SYMMETRY

Am Anfang war die Symmetrie – In the beginning was symmetry!

Werner Heisenberg, *Der Teil und das Ganze*:
Gespräche im Umkreis der Atomphysik, 1969,
English translation, *Physics and Beyond*, 1971.

by

Marc Bezem
Ulrik Buchholtz
Pierre Cagne
Bjørn Ian Dundas
Daniel R. Grayson

Book version: `8cfccd1` (2022-10-29)

This book is available at: https://unimath.github.io/SymmetryBook/book.pdf

To cite the book, the following BibTeX code may be useful:

```
@misc{Symmetry,
   title         = {Symmetry},
   author        = {Marc Bezem and Ulrik Buchholtz and Pierre Cagne
                    and Bjørn Ian Dundas and Daniel R. Grayson},
   date          = {2022-10-29},
   howpublished  = {\url{https://github.com/UniMath/SymmetryBook}},
   note          = {Commit: \texttt{8cfccd1}}
}
```

Short contents

Contents

ch::intro

1 Introduction to the topic of this book

Poincaré sagte gelegentlich, dass alle Mathematik eine Gruppengeschichte war. Ich erzählte ihm dann über dein Programm, das er nicht kannte.

Poincaré was saying that all of mathematics was a tale about groups. I then told him about your program, which he didn't know about.

(Letter from Sophus Lie to Felix Klein, October 1882)

This book is about symmetry and its many manifestations in mathematics. There are many kinds of symmetry and many ways of studying it. Euclidean plane geometry is the study of properties that are invariant under rigid motions of the plane. Other kinds of geometry arise by considering other notions of transformation. Univalent mathematics gives a new perspective on symmetries: Motions of the plane are forms of identifying the plane with itself in possibly non-trivial ways. It may also be useful to consider different presentations of planes (for instance as embedded in a common three-dimensional space) and different identifications between them. For instance, when drawing images in perspective we identify planes in the scene with the image plane, not in a rigid Euclidean way, but rather via a perspectivity (see Fig. ?). This gives rise to projective geometry.

Does that mean that a plane from the point of view of Euclidean geometry is not the same as a plane from the point of view of projective or affine geometry? Yes. These are of different types, because they have different notions of identification, and thus they have different properties.

Here we follow Quine's dictum: No entity without identity! To know a type of objects is to know what it means to identify representatives of the type. The collection of self-identifications (self-transformations) of a given object form a *group*.

Group theory emerged from many different directions in the latter half of the 19th century. Lagrange initiated the study of the invariants under permutations of the roots of a polynomial equation $f(x) = 0$, which culminated in the celebrated work of Abel and Galois. In number theory, Gauss had made detailed studies of modular arithmetic, proving for instance that the group of units of $\mathbb{Z}/p\mathbb{Z}$ is cyclic. Klein was bringing order to geometry by considering groups of transformation, while Lie was applying group theory in analysis to the study of differential equations.

Galois was the first to use the word “group” in a technical sense, speaking of collections of permutations closed under composition. He realized that the existence of resolvent equation is equivalent to the existence of a normal subgroup of prime index in the group of the equation.

Groupoids vs groups. The type of all squares in a euclidean plane form a groupoid. It is connected, because between any two there exist identifications between them. But there is no canonical identification.

When we say “the symmetry group of the square”, we can mean two things: 1) the symmetry group of a particular square; this is indeed a group, or 2) the connected groupoid of all squares; this is a “group up to conjugation”.

Vector spaces. Constructions and fields. Descartes and cartesian geometry.

Klein’s EP:

> Given a manifold and a transformation group acting on it, to investigate those properties of figures on that manifold that are invariant under transformations of that group.

and

> Given a manifold, and a transformation group acting on it, to study its *invariants*.

Invariant theory had previously been introduced in algebra and studied by Clebsch and Gordan.

(Mention continuity, differentiability, analyticity and Hilbert’s 5^{th} problem?)

Any finite automorphism group of the Riemann sphere is conjugate to a rotation group (automorphism group of the Euclidean sphere). [Dependency: diagonalizability] (Any complex representation of a finite group is conjugate to a unitary representation.)

All of mathematics is a tale, not about groups, but about ∞-groupoids. However, a lot of the action happens already with groups.

Glossary of coercions

MOVE TO BETTER PLACE Throughout this book we will use the following coercions to make the text more readable.

- If X is the pointed type (A, a), then $x : X$ means $x : A$.
- On hold, lacking context: If p and q are paths, then (p, q) means $(p, q)^=$.
- If e is a pair of a function and a proof, we also use e for the function.
- If e is an equivalence between types A and B, we use $\bar{e}$ for the identification of A and B induced by univalence.
- If $p : A = B$ with A and B types, then we use $\tilde{p}$ for the canonical equivalence from A to B (also only as function).
- If X is $(A, a, \ldots)$ with $a : A$, then pt_X and even just pt mean a.

How to read this book

...
A word of warning. We include a lot of figures to make it easier to follow the material. But like all mathematical writing, you'll get the most out of it, if you maintain a skeptical attitude: Do the pictures really accurately represent the formal constructions? Don't just believe us: Think about it!

The same goes for the proofs: When we say that something *clearly* follows, it should be *clear to you*. So clear, in fact, that you could go and convince a proof assistant, should you so desire.

Acknowledgement

The authors acknowledge the support of the Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters in Oslo, Norway, which funded and hosted the research project Homotopy Type Theory and Univalent Foundations during the academic year 2018/19.

2
An introduction to univalent mathematics

ch:univalent-mathematics

2.1 What is a type?

sec:what-is-a-type

In some computer programming languages, all variables are introduced along with a declaration of the type of thing they will refer to. Knowing the type of thing a variable refers to allows the computer to determine which expressions in the language are *grammatically well formed*[1], and hence valid. For example, if s is a string[2] and x is a real number, we may write $1/x$, but we may not write $1/s$.[3]

[1] The grammar of a programming language consists of all the language's rules. A statement or expression in a programming language is grammatically well formed if it follows all the rules.

[2] A *string* is a sequence of characters, such as "abcdefgh".

[3] In a programming language, the well formed expression $1/x$ may produce a run-time error if x happens to have the value 0.

To enable the programmer to express such declarations, names are introduced to refer to the various types of things. For example, the name Bool may be used to declare that a variable is a Boolean value[4], Int may refer to 32 bit integers, and Real may refer to 64 bit floating point numbers[5].

[4] A Boolean value is either *true* or *false*.

[5] An example of a *floating point number* is $.625 \times 2^{33}$ – the *mantissa* $.625$ and the *exponent* 33 are stored inside the floating point number. The "point", when the number is written in base 2 notation, is called "floating", because its position is easily changed by modifying the exponent.

Types occur in mathematics, too, and are used in the same way: all variables are introduced along with a declaration of the type of thing they will refer to. For example, one may say "consider a real number x", "consider a natural number n", "consider a point P of the plane", or "consider a line L of the plane". After that introduction, one may say that the *type* of n is *natural number* and that the *type* of P is *point of the plane*. Just as in a computer program, type declarations such as those are used to determine which mathematical statements are grammatically well formed. Thus one may write "P lies on L" or $1/x$, but not "L lies on P" nor $1/L$.[6]

[6] In mathematics there are no "run-time" errors; rather, it is legitimate to write the expression $1/x$ only if we already know that x is a non-zero real number.

Often ordinary English writing is good enough for such declarations in mathematics expositions, but, for convenience, mathematicians usually introduce symbolic names to refer to the various types of things under discussion. For example, the name $\mathbb{N}$ is usually used when declaring that a variable is a natural number, the name $\mathbb{Z}$ is usually used when declaring that a variable is an integer, and the name $\mathbb{R}$ is usually used when declaring that a variable is a real number. Ways are also given for constructing new type names from old ones: for example, the name $\mathbb{R} \times \mathbb{R}$ may be used when declaring that a variable is a point of the plane, for it conveys the information that a point of the plane is a pair of real numbers.

Once one becomes accustomed to the use of names such as $\mathbb{N}$ in mathematical writing and speaking, it is natural to take the next step and regard those names as denoting things that exist. Thus, we shall refer to $\mathbb{N}$ as the *type of all natural numbers*, and we will think of it as a mathematical object in its own right. Intuitively and informally, it is a collection whose members (or *elements*) are the natural numbers.

Once we view the various types as existing as mathematical objects, they become worthy of study. The language of mathematics is thereby improved, and the scope of mathematics is broadened. For example, we can consider statements such as "ℕ is infinite" and to try to prove it.

Historically, there was some hesitation[7] about introducing the collection of all natural numbers as a mathematical object, perhaps because if one were to attempt to build the collection from nothing by adding numbers to it one at a time, it would take an eternity to complete the assembly. We won't regard that as an obstacle.

[7] TO DO : Include some pointers to discussions of potential infinity and actual infinity, perhaps.

We have said that the types of things are used to determine whether mathematical statements are well formed. Therefore, if we expect "ℕ is infinite" to be a well-formed statement, we'll have to know what type of thing ℕ is, and we'll have to have a name for that type. Similarly, we'll have to know what type of thing that type is, and we'll have to have a name for it, and so on forever. Indeed, all of that is part of what will be presented in this chapter.

2.2 *Types, elements, families, and functions*

In this section we build on the intuition imparted in the previous section.

In *univalent mathematics*,[8] types are used to classify all mathematical objects. Every mathematical object is an *element* (or a *member*) of some (unique) *type*. Before one can talk about an object of a certain type, one must introduce the type itself. There are enough ways to form new types from old ones to provide everything we need to write mathematics.

[8] The term "univalent" is a word coined by Vladimir Voevodsky, who introduced it to describe his principle that types that are *equivalent* in a certain sense can be identified with each other. The principle is stated precisely in Principle 2.13.2. As Voevodsky explained, the word comes from a Russian translation of a mathematics book, where the English mathematical term "faithful" was translated into Russian as the Russian word that sounds like "univalent". He also said "Indeed these foundations seem to be faithful to the way in which I think about mathematical objects in my head."

One expresses the declaration that an object a is an element of the *type* X by writing $a : X$.[9]

[9] The notation in mathematics based on *set theory* that corresponds (sort of) to this is $a \in X$.

Using that notation, each variable x is introduced along with a declaration of the form $x : X$, which declares that x will refer to something of type X, but provides no other information about x. The declared types of the variables are used to determine which statements of the theory are grammatically well formed.

After introducing a variable $x : X$, it may be possible to form an expression T representing a type, all of whose components have been already been given a meaning. (Here the variable x is regarded also as having already been given a meaning, even though the only thing known about it is its type.) To clarify the dependence of T on x primarily, we may write $T(x)$ (or T_x) instead of T. Such an expression will be called a *family of types* parametrized by the variable x of type X. Such a family provides a variety of types, for, if a is any expression denoting an object of X, one may replace all occurrences of x by a in T, thereby obtaining a new expression representing a type, which may be regarded as a member of the family, and which may be denoted by $T(a)$.

Naturally, if the expression T doesn't actually involve the variable x, then the members of the family are all the same, and we'll refer to the family as a *constant family* of types.

Here's an example of a family of types: we let n be a natural number and P_n be the type of n-sided polygons in the plane. It gives a family of types parametrized by the natural numbers.[10] One of the members of the family is the type P_5 of all pentagons in the plane.

[10] Well, either we should suppose $n \geq 3$, or make some other stipulation about P_n for $n < 3$.

A family of types may be parametrized by more than one variable. For example, after introducing a variable $x : X$ and a family of types T parametrized by x, we may introduce a variable $t : T$. Then it may be possible to form an expression S representing a type that involves the variables x and t. Such an expression will be called a family of types parametrized by x and t, and we may write $S(x, t)$ instead of S to emphasize the dependence on x and t. The same sort of thing works with more variables.

After introducing a variable $x : X$ and a family of types T, it may be possible to form an expression e of type T, all of whose components have been already been given a meaning. Such an expression will also be called a *family of elements of* T parametrized by the elements of X, when we wish to focus on the dependence of e (and perhaps T) on the variable x. To clarify the dependence of e on x primarily, we may write $e(x)$ (or e_x) instead of e. Such a family provides a variety of elements of T, for, if a is any expression denoting an object of X, one may replace all occurrences of x by a in e and in T, thereby obtaining an element of $T(a)$, which may be regarded as a *member* of the family and which will be denoted by $e(a)$.

Naturally, if the expressions e and T don't actually involve the variable x, then the members of the family are all the same, and we'll refer to the family as a *constant family* of elements.

Here's an example of a family of elements in a constant family of types: we let n be a natural number and consider the real number $\sqrt{n}$. It gives a family of real numbers parametrized by the natural numbers. (The family may also be called a *sequence* of real numbers). One of the members of the family is $\sqrt{11}$.

Here's an example of a family of elements in a (non-constant) family of types: we let n be a natural number and P_n be the type of n-sided polygons in the plane, as we did above. Then we consider the regular n-sided polygon p_n of radius 1 with a vertex on the positive x-axis. We see that $p_n : P_n$. One of the members of the family is the regular pentagon $p_5 : P_5$ of radius 1 with a vertex on the positive x-axis.

The type X containing the variable for a family of types or a family of elements is called the *parameter type* of the family.

Just as a family of types may depend on more than one variable, a family of elements may also depend on more than one variable.

Families of elements can be enclosed in mathematical objects called *functions* (or *maps*), as one might expect. Let e be a family of elements of a family of types T, both of which are parametrized by the elements x of X. We use the notation $x \mapsto e$ for the function that sends an element a of X to the element $e(a)$ of $T(a)$; the notation $x \mapsto e$ can be read as "x maps to e" or "x goes to e". (Recall that $e(a)$ is the expression that is obtained from e by replacing all occurrences of x in e by a.) If we name the function f, then that element of T will be denoted by $f(a)$. The *type* of the function $x \mapsto e$ is called a *product type* and will be denoted by $\prod_{x:X} T$; if T is a constant family of types, then the type will also be called a *function type* and will be denoted by $X \to T$. Thus when we write $f : X \to T$, we mean that f is an element of the type $X \to T$, and we are saying that f is a function from X to T. The type X may be called the *domain* of f, and the type T may be called the *codomain* of f.

An example of a function is the function $n \mapsto \sqrt{n}$ of type $\mathbb{N} \to \mathbb{R}$.

Another example of a function is the function $n \mapsto p_n$ of type $\prod_{n:\mathbb{N}} P_n$, where P_n is the type of polygons introduced above, and p_n is the polygon introduced above.

Another example of a function is the function $m \mapsto (n \mapsto m + n)$ of type $\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$. It is a function that accepts a natural number as argument and returns a function as its value. The function returned is of type $\mathbb{N} \to \mathbb{N}$. It accepts a natural number as argument and returns a natural number as value.

The reader may wonder why the word "product" is used when speaking of product types. To motivate that, we consider a simple example informally. We take X to be a type with just two elements, b and c. We take $T(x)$ to be a family of types parametrized by the elements of X, with $T(b)$ being a type with 5 elements and $T(c)$ being a type with 11 elements. Then the various functions f of type $\prod_{x:X} T(x)$ are plausibly obtained by picking a suitable element for $f(b)$ from the 5 possibilities in $T(b)$ and by picking a suitable element for $f(c)$ from the 11 possibilities in $T(c)$. The number of ways to make both choices is 5×11, which is a *product* of two numbers. Thus $\prod_{x:X} T(x)$ is sort of like the product of $T(b)$ and $T(c)$, at least as far as counting is concerned.

The reader may wonder why we bother with functions at all: doesn't the expression e serve just as well as the function $x \mapsto e$, for all practical purposes? The answer is no. One reason is that the expression e doesn't inform the reader that the variable under consideration is x. Another reason is that we may want to use the variable x for elements of a different type later on: then $e(x)$ is no longer well formed. For example, imagine first writing this: "For a natural number n we consider the real number $\sqrt{n}$" and then writing this: "Now consider a triangle n in the plane." The result is that $\sqrt{n}$ is no longer usable, whereas the function $n \mapsto \sqrt{n}$ has enclosed the variable and the family into a single object and remains usable.[11]

Once a family e has been enclosed in the function $x \mapsto e$, the variable x is referred to as a *dummy variable* or as a *bound variable*.[12] This signifies that the name of the variable no longer matters, in other words, that $x \mapsto e(x)$ and $t \mapsto e(t)$ may regarded as identical. Moreover, the variable x that occurs inside the function $x \mapsto e$ is regarded as unrelated to variables x which may appear elsewhere in the discussion.

[11]Students of trigonometry are already familiar with the concept of function, as something enclosed this way. The sine and cosine functions, sin and cos, are examples.

[12]Students of calculus are familiar with the concept of dummy variable and are accustomed to using identities such as $\int_a^b f(t)\,dt = \int_a^b f(x)\,dx$.

If the variable x in our notation $x \mapsto e(x)$ is a dummy variable, and its name doesn't matter, then we may consider the possibility of not specifying a variable at all. We introduce now a methodical way to do that, by replacing the occurrences of the variable x in the expression $e(x)$ by an *underscore*, yielding $e(\_)$ as alternative notation for the function $x \mapsto e(x)$. For example, the notation $\sqrt{\_}$ can serve as alternative notation for the function $n \mapsto \sqrt{n}$ introduced above, and $2 + \_$ can serve as alternative notation for the function $n \mapsto 2 + n$ of type $\mathbb{N} \to \mathbb{N}$.

We have mentioned above the possibility of giving a name to a function. We expand on that now by introducing notation for making and for using *definitions*.

The notation $x :\equiv z$ will be an announcement that we are defining the expression x to be the expression z, all of whose components have already been given a meaning; in that case, we will say that x has been *defined* to be (or to mean) z. The forms allowed for the expression x will be made clear by the examples we give.

For example, after writing $n :\equiv 12$, we will say that n has been defined to be 12.

For another example, the function f that we named above may be introduced by writing $f :\equiv (x \mapsto e(x))$. Alternatively and more traditionally, we may write $f(x) :\equiv e(x)$.

The notation $b \equiv c$ will denote the statement that the expressions b and c become the same thing if all the subexpressions within b or c are expanded according to their definitions, if any; in that case, we will say that b and c are *the same by definition*. For example, after writing $n :\equiv 12$ and $m :\equiv n$, we may say that $j + 12 \equiv j + m$ and that $m \times 11 \equiv 12 \times 11$.

Whenever two expressions are the same by definition, we may replace one with the other inside any other expression, because the expansion of definitions is regarded as trivial and transparent.

We proceed now to the promised example. Consider functions $f : X \to Y$ and $g : Y \to Z$. We define the *composite* function $g \circ f : X \to Z$ by setting $g \circ f :\equiv (a \mapsto g(f(a)))$. In other words, it is the function that sends an arbitrary element a of X to $g(f(a))$ in Z. (The expression $g \circ f$ may be read as "g circle f" or as "g composed with f".) The composite function $g \circ f$ may also be denoted simply by gf.

Now consider functions $f : X \to Y$, $g : Y \to Z$, and $h : Z \to W$. Then $(h \circ g) \circ f$ and $h \circ (g \circ f)$ are the same by definition, since applying the definitions within expands both expressions to $a \mapsto h(g(f(a)))$. In other words, we have established that $(h \circ g) \circ f \equiv h \circ (g \circ f)$. Thus, we may write $h \circ g \circ f$ for either expression, without danger of confusion.

One may define the identity function $\mathrm{id}_X : X \to X$ by setting $\mathrm{id}_X :\equiv (a \mapsto a)$. Application of definitions shows that $f \circ \mathrm{id}_X$ is the same by definition as $a \mapsto f(a)$, which, by a standard convention, which we adopt[13], is to be regarded as the same as f. In other words, we have established that $f \circ \mathrm{id}_X \equiv f$. A similar computation applies to $\mathrm{id}_Y \circ f$.

In the following sections we will present various other elementary types and elementary ways to make new types from old ones.

page:idofetaf

2.3 *Universes*

sec:universes

In Section 2.2 we have introduced the objects known as *types*. They have *elements*, and the type an element belongs to determines the type of thing that it is. At various points in the sequel, it will be convenient for types also to be elements, for that will allow us, for example, to define families of types just as easily as we define families of elements. To achieve this convenience, we introduce types that are *universes*. Some care is required, for the first temptation is to posit a single new type $\mathcal{U}$ called *the universe*, so that every type is realized as an element of $\mathcal{U}$. This universe would be "the type of all types", but introducing it would lead to an absurdity, for roughly the same reason that introduction of a "set of all sets" leads to the absurdity in traditional mathematics known as Russell's paradox.[14] Some later approaches to set theory included the notion of a *class*, with the collection of all sets being the primary example of a class. Classes are much like sets, and every set is a class, but not every class is a set. Then

[13]The convention that $f \equiv (a \mapsto f(a))$ is referred to as the *η-rule* in the jargon of type theory.

[14]In fact, type theory can trace its origins to Russell's paradox, announced in a 1902 letter to Frege as follows:

> There is just one point where I have encountered a difficulty. You state that a function too, can act as the indeterminate element. This I formerly believed, but now this view seems doubtful to me because of the following contradiction. Let w be the predicate: to be a predicate that cannot be predicated of itself. Can w be predicated of itself? From each answer its opposite follows. Therefore we must conclude that w is not a predicate. Likewise there is no class (as a totality) of those classes which, each taken as a totality, do not belong to themselves.

To which Frege replied:

> Incidentally, it seems to me that the expression "a predicate is predicated of itself" is not exact. A predicate is as a rule a first-level function, and this function requires an object as argument and cannot have itself as argument (subject).

Russell then quickly added *Appendex B* to his *Principles of Mathematics* (1903), in which he said that "it is the distinction of logical types that is the key to the whole mystery", where types are the *ranges of significance* of variables. For more on the history of type theory, see Coquand[15].

[15]Thierry Coquand. "Type Theory". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Metaphysics Research Lab, Stanford University, 2018. URL: https://plato.stanford.edu/archives/fall2018/entries/type-theory/.

one may wonder what sort of thing the collection of all classes would be. Such musings are resolved in univalent mathematics as follows.

(1) There are some types called *universes*.

(2) If $\mathcal{U}$ is a universe, and $X : \mathcal{U}$ is an element of $\mathcal{U}$, then X is a type.

(3) If X is a type, then it appears as an element in some universe $\mathcal{U}$. Moreover, if X and Y are types, then there is a universe $\mathcal{U}$ containing both of them.

(4) If $\mathcal{U}$ and $\mathcal{U}'$ are universes, $\mathcal{U} : \mathcal{U}'$, X is a type, and $X : \mathcal{U}$, then also $X : \mathcal{U}'$. (Thus we may regard $\mathcal{U}'$ as being *larger* than $\mathcal{U}$.)

(5) There is a particular universe $\mathcal{U}_0$, which we single out to serve as a repository for certain basic types to be introduced in the sequel. Moreover, $\mathcal{U}_0 : \mathcal{U}$ for every other universe $\mathcal{U}$, and thus $\mathcal{U}_0$ is the *smallest* universe.

It follows from the properties above that there are an infinite number of universes, for each one is an element of a larger one.

Now suppose we have a type X and a family $T(x)$ of types parametrized by a variable x of type X. Choose a universe U with $T(x) : U$. Then we can make a function of type $X \to U$, namely $f :\equiv (x \mapsto T(x))$. Conversely, if f' is a function of type $X \to \mathcal{U}$, then we can make a family of types parametrized by x, namely $T' :\equiv f'(x)$. The flexibility offered by this correspondence between families of types in $\mathcal{U}$ and functions to $\mathcal{U}$ will often be used.

2.4 *The type of natural numbers*

sec:natural-numbers

Here are Peano's rules[16] for constructing the natural numbers in the form that is used in type theory.

[16] Giuseppe Peano. *Arithmetices principia: nova methodo*. See also https://github.com/mdnahas/Peano_Book/ for a parallel translation by Vincent Verheyen. Fratres Bocca, 1889. URL: https://books.google.com/books?id=z80GAAAAYAAJ.

P1 (P1) there is a type called $\mathbb{N}$ in the universe $\mathcal{U}_0$ (whose elements will be called *natural numbers*);

P2 (P2) there is an element of $\mathbb{N}$ called 0, called *zero*;

P3 (P3) if m is a natural number, then there is also a natural number $\mathrm{succ}(m)$, called the *successor* of m;

P4 (P4) suppose we are given:

a) a family of types $X(m)$ parametrized by a variable m of type $\mathbb{N}$;

b) an element a of $X(0)$; and

c) a family of functions $g_m : X(m) \to X(\mathrm{succ}(m))$.

Then from those data we are provided with a family of elements $f(m) : X(m)$.

The first three rules present few problems for the reader. They provide us with the smallest natural number $0 : \mathbb{N}$, and we may introduce as many others as we like with the following definitions.

$$
\begin{aligned}
1 &:\equiv \mathrm{succ}(0) \\
2 &:\equiv \mathrm{succ}(1) \\
3 &:\equiv \mathrm{succ}(2) \\
&\vdots
\end{aligned}
$$

You may recognize rule (P4) as "the principle of mathematical induction" or as "defining a function by recursion".[17] We will refer to it simply as "induction on $\mathbb{N}$". The resulting family f may be regarded as having been defined inductively by the two declarations $f(0) :\equiv a$ and $f(\mathrm{succ}(m)) :\equiv g_m(f(m))$, and indeed, we will often simply write such a pair of declarations as a shorthand way of applying rule (P4). The two declarations cover the two ways of introducing elements of $\mathbb{N}$ via the use of the two rules (P2) and (P3). (In terms of computer programming, those two declarations amount to the code for a recursive subroutine that can handle any incoming natural number.)

[17]Rule (P4) and our logical framework are stronger than in Peano's original formulation, and this allows us to omit some rules that Peano had to include: that different natural numbers have different successors; and that no number has 0 as its successor. Those omitted rules remain true in this formulation and can be proved from the other rules, after we have introduced the notion of equality in our logical framework.

With that notation in hand, speaking informally, we may regard (P4) above as defining the family f by the following infinite sequence of definitions.

$$
\begin{aligned}
f(0) &:\equiv a \\
f(1) &:\equiv g_0(a) \\
f(2) &:\equiv g_1(g_0(a)) \\
f(3) &:\equiv g_2(g_1(g_0(a))) \\
&\vdots
\end{aligned}
$$

(The need for the rule (P4) arises from our inability to write down an infinite sequence of definitions in a finite amount of space, and from the need for $f(m)$ to be defined when m is a variable of type $\mathbb{N}$, and thus is not known to be equal to 0, nor to 1, nor to 2, etc.)

We may use induction on $\mathbb{N}$ to define of *iteration* of functions. Let Y be a type, and suppose we have a function $e : Y \to Y$. We define by induction on $\mathbb{N}$ the m-fold *iteration* $e^m : Y \to Y$ by setting $e^0 :\equiv \mathrm{id}_Y$ and $e^{\mathrm{succ}(m)} :\equiv e \circ e^m$. (Here we apply rule (P4) with the the type $Y \to Y$ as the family of types $X(m)$, the identity function id_Y for a, and the function $d \mapsto e \circ d$ for the family $g_m : (Y \to Y) \to (Y \to Y)$ of functions.)

We may now define addition of natural numbers by induction on $\mathbb{N}$. For natural numbers n and m we define $n + m : \mathbb{N}$ by induction on $\mathbb{N}$ with respect to the variable m by setting $n + 0 :\equiv n$ and $n + \mathrm{succ}(m) :\equiv \mathrm{succ}(n + m)$. (The reader should be able to extract the family $X(m)$, the element a, and the family of functions g_m from that pair of definitions.) Application of definitions shows, for example, that $2 + 2$ and 4 are the same by definition, and thus we may write $2 + 2 \equiv 4$, because both expressions reduce to $\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(\mathrm{succ}(0))))$.

Similarly we define the product $m \cdot n : \mathbb{N}$ by induction on m by setting setting $0 \cdot n :\equiv 0$ and $\mathrm{succ}(m) \cdot n :\equiv (m \cdot n) + n$.

Alternatively (and equivalently) we may use iteration of functions to define addition and multiplication, by setting $n + m :\equiv \mathrm{succ}^m(n)$ and $m \cdot n :\equiv (i \mapsto i + n)^m(0)$.

Finally, we may define the factorial function $\mathrm{fact} : \mathbb{N} \to \mathbb{N}$ by induction on $\mathbb{N}$, setting $\mathrm{fact}(0) :\equiv 1$ and $\mathrm{fact}(\mathrm{succ}(m)) :\equiv \mathrm{succ}(m) \cdot \mathrm{fact}(m)$. (One can see that this definition applies rule (P4) with $X(m) :\equiv \mathbb{N}$, with 1 for a, and with the function $n \mapsto \mathrm{succ}(m) \cdot n$ for g_m.) Application of the definitions shows, for example, that $\mathrm{fact}(3) \equiv 6$, as the reader may verify.

2.5 *Identity types*

sec:identity-types

One of the most important types is the *identity type*, which implements the intuitive notion of equality. Identity between two elements may be considered only when the two elements are of the same type; we shall have no need to compare elements of different types.

Here are the rules for constructing and using identity types.

rules-for-equality

E1 (E1) for any type X and for any elements a and b of it, there is a *identity type* $a \overset{=}{\to} b$; moreover, if X is an element of a universe $\mathcal{U}$, then so is $a \overset{=}{\to} b$.

E2 (E2) for any type X and for any element a of it, there is an element refl_a of type $a \overset{=}{\to} a$ (the name refl comes from the word “reflexivity”)

E3 (E3) suppose we are given:

a) a type X and an element $a : X$;

b) a family of types $P(b, e, \ldots)$ parametrized by a variable b of type X, a variable e of type $a \overset{=}{\to} b$, and perhaps some further variables; and

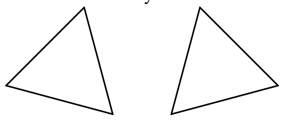
c) an element p of $P(a, \mathrm{refl}_a, \ldots)$.

Then from those data we are provided with a family of elements $f(b, e, \ldots) : P(b, e, \ldots)$. Moreover, $f(a, \mathrm{refl}_a, \ldots) \equiv p$.

We will refer to an element i of $a \overset{=}{\to} b$ as an *identification* of a with b, or simply as an *identity*. Since the word “identification” is a long one, we may also refer to i as a *path* from a to b – this has the advantage of incorporating the intuition that an identification may proceed gradually through intermediate steps.

The need to record, using the element i, the way we identify a with b may come as a surprise, since normally, in mathematics, one is accustomed to regarding a as either equal to b or not. However, this reflects a situation commonly encountered in geometry when *congruence* of geometric figures is considered. For example, in Euclidean space, two equilateral triangles of the same size are congruent in six (different) ways.[18] The chief novelty of univalent mathematics is that the basic logical notion of equality, as implemented by the identity types $a \overset{=}{\to} b$, is carefully engineered to accommodate notions of congruence and symmetry from diverse areas of mathematics, including geometry. Exposing that point of view in the context of geometry is the main point of this book.

[18]Six, since we allow reflections, otherwise there are only three.



In light of the analogy with geometry just introduced, we will refer to an element i of $a \overset{=}{\to} a$ as a *symmetry* of a. Think, for example, of a congruence of a triangle with itself. An example of a non-trivial symmetry will be seen in Exercise 2.13.3.

Consider the identity type $\mathrm{fact}(2) \xrightarrow{=} 2$, where fact denotes the factorial function defined in Section 2.4. Expansion of the definitions in the equation $\mathrm{fact}(2) \xrightarrow{=} 2$ simplifies it to $\mathrm{succ}(\mathrm{succ}(0)) \xrightarrow{=} \mathrm{succ}(\mathrm{succ}(0))$, so we see from rule (E2) that $\mathrm{refl}_{\mathrm{succ}(\mathrm{succ}(0))}$ serves as an element of it.[19] We may also write either refl_2 or $\mathrm{refl}_{\mathrm{fact}(2)}$ for that element. A student might want a more detailed derivation that fact(2) may be identified with 2, but as a result of our convention above that definitions may be applied without changing anything, the application of definitions, including inductive definitions, is normally regarded as a trivial operation, and the details are usually omitted.

[19]We will see later that numbers don't have non-trivial symmetries, as one would expect, so the possibility that there are other ways to identify fact(2) with 2 doesn't arise.

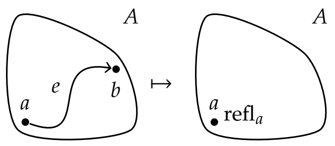
We will refer to rule (E3) as "induction for identity". To signal that we wish to apply it, we may announce that we argue *by induction on e*.

The family f resulting from an application of rule (E3) may be regarded as having been completely defined by the single declaration $f(a, \mathrm{refl}_a) :\equiv p$, and indeed, we will often simply write such a declaration as a shorthand way of applying rule (E3). The rule says that to construct something from every identification e of a with something else, it suffices to consider the special case where the identification e is $\mathrm{refl}_a : a \xrightarrow{=} a$.[20]

[20]Notice that the single special case in such an induction corresponds to the single way of introducing elements of identity types via rule (E2), and compare that with (P4), which dealt with the two ways of introducing elements of $\mathbb{N}$.

Intuitively, the induction principle for identity amounts to saying that the element refl_a "generates" the system of types $a \xrightarrow{=} b$, as b ranges over elements of A.[21]

[21]We can also use a geometric intuition: when b "freely ranges" over elements of A, together with a path $e : a \xrightarrow{=} b$, while we keep the element a fixed, we can picture e as a piece of string winding through A, and the "freeness" of the pair (b, e) allows us to pull the string e, and b with it, until we have the constant path at a, refl_a.



Conversely, we can imagine b starting at a and e starting out as refl_a, and then think of b roaming throughout A, pulling the string e along with it, until it finds every path from a to some other element.

Equality is *symmetric*, in the sense that an identification of a with b may be reversed to give an identification of b with a. In order to produce an element of $b \xrightarrow{=} a$ from an element e of $a \xrightarrow{=} b$, for any b and e, we argue by induction. We let $P(b, e)$ be $b \xrightarrow{=} a$ for any b of type X and for any e of type $a \xrightarrow{=} b$, for use in rule (E3) above. Application of rule (E3) reduces us to the case where b is a and p is refl_a, and our task is now to produce an element of $a \xrightarrow{=} a$; we choose refl_a for it.

Equality is also *transitive*, and is established the same way. For each $a, b, c : X$ and for each $p : a \xrightarrow{=} b$ and for each $q : b \xrightarrow{=} c$ we want to produce an element of type $a \xrightarrow{=} c$. By induction on q we are reduced to the case where c is b and q is refl_b, and we are to produce an element of $a \xrightarrow{=} b$. The element p serves the purpose.

Now we state our symmetry result a little more formally.

def:eq-symm

DEFINITION 2.5.1. For any type X and for any $a, b : X$, let

$$\mathrm{symm}_{a,b} : (a \xrightarrow{=} b) \to (b \xrightarrow{=} a)$$

be the function defined by induction by setting $\mathrm{symm}_{a,a}(\mathrm{refl}_a) :\equiv \mathrm{refl}_a$.

This operation on paths is called *path inverse*, and we may abbreviate $\mathrm{symm}_{a,b}(p)$ as p^{-1}. ⌟

Similarly, we formulate transitivity a little more formally, as follows.

def:eq-trans

DEFINITION 2.5.2. For any type X and for any $a, b, c : X$, let

$$\mathrm{trans}_{a,b,c} : (a \xrightarrow{=} b) \to ((b \xrightarrow{=} c) \to (a \xrightarrow{=} c))$$

be the function defined by induction by setting $(\mathrm{trans}_{a,b,b}(p))(\mathrm{refl}_b) :\equiv p$.

This binary operation is called *path composition* or *path concatenation*, and we may abbreviate $(\mathrm{trans}_{a,b,c}(p))(q)$ as either $p * q$, or as $q \cdot p$, qp, or $q \circ p$. ⌟

The intuition that the path p summarizes a gradual change from a to b, and q summarizes a gradual change from b to c, leads to the intuition that $p * q$ progresses gradually from a to c by first changing a to b and then changing b to c; see Fig. 2.1.

The notation $q \circ p$ for path composition, with p and q in reverse order, fits our intution particularly well when the paths are related to functions and the composition of the paths is related to the composition of the related functions in the same order, as happens, for example, in connection with *transport* (defined below in Definition 2.5.4) in Exercise 2.5.5.

The types of $\mathrm{symm}_{a,b}$ and $\mathrm{trans}_{a,b,c}$ express that identity is symmetric and transitive. Another view of $\mathrm{symm}_{a,b}$ and $\mathrm{trans}_{a,b,c}$ is that they are operations on identifications, namely reversing an identification and concatenating two identifications. The results of various combinations of these operations can often be identified: we formulate some of these identifications in the following exercise.

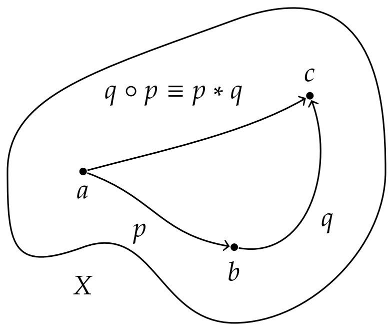


FIGURE 2.1: Composition (also called concatenation) of paths in X

fig:path-concatenation

xca:path-groupoid-laws

EXERCISE 2.5.3. Let X be a type and let $a, b, c, d : X$ be elements.

(1) For $p : a \xrightarrow{=} b$, construct an identification of type $p * \mathrm{refl}_b \xrightarrow{=} p$.

(2) For $p : a \xrightarrow{=} b$, construct an identification of type $\mathrm{refl}_a * p \xrightarrow{=} p$.

(3) For $p : a \xrightarrow{=} b$, $q : b \xrightarrow{=} c$, and $r : c \xrightarrow{=} d$, construct an identification of type $(p * q) * r \xrightarrow{=} p * (q * r)$.

(4) For $p : a \xrightarrow{=} b$, construct an identification of type $p^{-1} * p \xrightarrow{=} \mathrm{refl}_b$.

(5) For $p : a \xrightarrow{=} b$, construct an identification of type $p * p^{-1} \xrightarrow{=} \mathrm{refl}_a$.

(6) For $p : a \xrightarrow{=} b$, construct an identification of type $(p^{-1})^{-1} \xrightarrow{=} p$. ⌟

Given an element $p : a \xrightarrow{=} a$, we may use concatenation to define powers $p^n : a \xrightarrow{=} a$ by induction on $n : \mathbb{N}$; we set $p^0 :\equiv \mathrm{refl}_a$ and $p^{n+1} :\equiv p \cdot p^n$. Negative powers p^{-n} are defined as $(p^{-1})^n$.[22]

[22] We haven't yet assigned a meaning to $-n$, but after we introduce the set of integers Z below in Definition 3.2.1, we'll be justified in writing p^z for any $z : \mathrm{Z}$. See also Example 2.12.9.

One frequent use of elements of identity types is in *substitution*, which is the logical principle that supports our intuition that when x can by identified with y, we may replace x by y in mathematical expressions at will. A wrinkle new to students will likely be that, in our logical framework where there may be various ways to identify x with y, one must specify the identification used in the substitution. Thus one may prefer to speak of using an identification to *transport* properties and data about x to properties and data about y.

Here is a geometric example: if x is a triangle of area 3 in the plane, and y is congruent to x, then y also has area 3.

Here is another example: if x is a right triangle in the plane, and y is congruent to x, then y is also a right triangle, and the congruence informs us which of the 3 angles of y is the right angle.

Now we introduce the notion more formally.

Let X be a type, and let $T(x)$ be a family of types parametrized by a variable $x : X$ (as discussed in Section 2.2). Suppose $a, b : X$ and $e : a \xrightarrow{=} b$. Then we may construct a function of type $T(a) \to T(b)$. We define one specific such function by induction on e, by taking its value on refl_a of type $a \xrightarrow{=} a$ to be the identity function on $T(a)$. We record that definition as follows.

def:transport

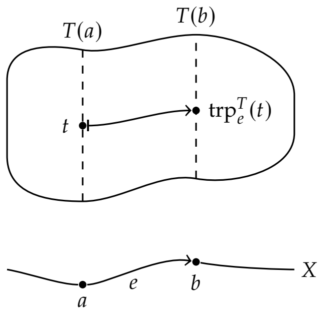
DEFINITION 2.5.4. The function

$$\mathrm{trp}^T_e : T(a) \to T(b)$$

is defined by induction setting $\mathrm{trp}^T_{\mathrm{refl}_a} :\equiv \mathrm{id}_{T(a)}$. ⌟

The function thus defined may be called *the transport function in the type family T along the path e*, or, less verbosely, *transport*.[23] We may also simplify the notation to just trp_e. The transport functions behave as expected: we may construct an identification of type $\mathrm{trp}_{e'\circ e} \xrightarrow{=} \mathrm{trp}_{e'} \circ \mathrm{trp}_e$. In words: transport along the composition $e \circ e'$ can be identified with the composition of the two transport functions. This may be proved by induction in the following exercise.

[23] We sometimes picture this schematically as follows: We draw X as a (mostly horizontal) line, and we draw each type $T(x)$ as a vertical line lying over $x : X$. As x moves around in X, these lines can change shape, and taken all together they form a 2-dimensional blob lying over X. The transport functions map points between the vertical lines.



xca:trp-compose

EXERCISE 2.5.5. Let X be a type, and let $T(x)$ be a family of types parametrized by a variable $x : X$. Suppose we are given elements $a, b, c : X$, $e : a \xrightarrow{=} b$, and $e' : b \xrightarrow{=} c$. Construct an identification of type

$$\mathrm{trp}_{e'\circ e} \xrightarrow{=} \mathrm{trp}_{e'} \circ \mathrm{trp}_e. \quad ⌟$$

Yet another example of good behavior is given in the following exercise.

xca:trp-nondep

EXERCISE 2.5.6. Let X, Y be types. As discussed in Section 2.2, we may regard the expression Y as a constant family of types parametrized by a variable $x : X$. Produce an identification of type $\mathrm{trp}^Y_p \xrightarrow{=} \mathrm{id}_Y$, for any path $p : a \xrightarrow{=} b$. ⌟

In Section 2.15 below we will discuss what it means for a type to have at most one element. When the types $T(x)$ may have more than one element, we may regard an element of $T(x)$ as providing additional *structure* on x. In that case, we will refer to the transport function $\mathrm{trp}_e : T(a) \to T(b)$ as *transport of structure* from a to b.

Take, for example, $T(x) :\equiv (x \xrightarrow{=} x)$. Then trp_e is of type $(a \xrightarrow{=} a) \to (b \xrightarrow{=} b)$ and transports a symmetry of a to a symmetry of b.

By contrast, when the types $T(x)$ have at most one element, we may regard an element of $T(x)$ as providing a proof of a property of x. In that case, the transport function $\mathrm{trp}_e : T(a) \to T(b)$ provides a way to establish a claim about b from a claim about a, so we will refer to it as *substitution*. In other words, elements that can be identified have the same properties.

2.6 *Product types*

sec:product-types

Functions and product types have been introduced in Section 2.2, where we have also explained how to create a function by enclosing a family of elements in one. In this section we treat functions and product types in more detail.

Recall that if X is a type and $Y(x)$ is a family of types parametrized by a variable x of type X, then there is a *product type* $\prod_{x:X} Y(x)$ whose elements f are functions that provide elements $f(a)$ of type $Y(a)$, one for each $a : X$. We will refer to X as the *parameter type* of the product. By contrast, if Y happens to be a constant family of types, then $\prod_{x:X} Y$ will also be denoted by $X \to Y$, and it will also be called a *function type*.

If X and $Y(x)$ are elements of a universe $\mathcal{U}$, then so is $\prod_{x:X} Y(x)$.

Functions preserve identity, and we will use this frequently later on. More precisely, functions induce maps on identity types, as the following definition makes precise.

def:ap

Definition 2.6.1. For all types X, Y, functions $f : X \to Y$ and elements $x, x' : X$, the function

$$\mathrm{ap}_{f,x,x'} : (x \overset{=}{\to} x') \to (f(x) \overset{=}{\to} f(x'))$$

is defined by induction by setting $\mathrm{ap}_{f,x,x}(\mathrm{refl}_x) :\equiv \mathrm{refl}_{f(x)}$. ⌟

The function $\mathrm{ap}_{f,x,x'}$, for any elements x and x' of X, is called an *application* of f to paths or to identities, and this explains the choice of the symbol ap in the notation for it. It may also be called the function (or map) *induced* by f on identity types.

When x and x' are clear from the context, we may abbreviate $\mathrm{ap}_{f,x,x'}$ by writing ap_f instead. For convenience, we may abbreviate it even further, writing $f(p)$ for $\mathrm{ap}_f(p)$.

The following lemma shows that ap_f is compatible with composition.

lem:apcomp

Lemma 2.6.2. *Given a function $f : X \to Y$, and elements $x, x', x'' : X$, and paths $p : x \overset{=}{\to} x'$ and $p' : x' \overset{=}{\to} x''$, we may construct an identification of type $\mathrm{ap}_f(p' \cdot p) \overset{=}{\to} \mathrm{ap}_f(p') \cdot \mathrm{ap}_f(p)$.*

Proof. By induction on p and p', one reduces to producing an identification of type

$$\mathrm{ap}_f(\mathrm{refl}_x \cdot \mathrm{refl}_x) \overset{=}{\to} \mathrm{ap}_f(\mathrm{refl}_x) \cdot \mathrm{ap}_f(\mathrm{refl}_x).$$

Both sides of the equation are equal to $\mathrm{refl}_{f(x)}$ by definition, so the element $\mathrm{refl}_{\mathrm{refl}_{f(x)}}$ has that type. □

In a similar way one shows that ap_f is compatible with path inverse, by constructing an identity of type $\mathrm{ap}_f(p^{-1}) \overset{=}{\to} (\mathrm{ap}_f(p))^{-1}$. One may also construct an identity of type $\mathrm{ap}_{\mathrm{id}}(p) \overset{=}{\to} p$.

xca:trp-ap

Exercise 2.6.3. Let X be a type, and let $T(x)$ be a family of types parametrized by a variable $x : X$. Furthermore, let A be a type, let $f : A \to X$ be a function, let a and a' be elements of A, and let $p : a \overset{=}{\to} a'$ be a path. Verify that the two functions $\mathrm{trp}_p^{T\circ f}$ and $\mathrm{trp}^T_{\mathrm{ap}_f(p)}$ are of type $T(f(a)) \to T(f(a'))$. Then construct an identification between them, i.e., construct an element of type $\mathrm{trp}_p^{T\circ f} \overset{=}{\to} \mathrm{trp}^T_{\mathrm{ap}_f(p)}$. ⌟

If two functions f and g of type $\prod_{x:X} Y(x)$ can be identified, then their values can be identified, i.e., for every element x of X, we may produce an identification of type $f(x) \overset{=}{\to} g(x)$, which can be constructed by induction, as follows.

def:ptw

Definition 2.6.4. Let $f, g : \prod_{x:X} Y(x)$. Define the function

$$\mathrm{ptw}_{f,g} : (f \overset{=}{\to} g) \to \left(\prod_{x:X} f(x) \overset{=}{\to} g(x) \right),$$

by induction by setting $\mathrm{ptw}_{f,f}(\mathrm{refl}_f) :\equiv x \mapsto \mathrm{refl}_{f(x)}$.[24] ⌟

[24]The notation ptw is chosen to remind the reader of the word "point-wise", because the identities are provided just for each point x. An alternative approach goes by considering, for any $x : X$, the evaluation function $\mathrm{ev}_x : (\prod_{x:X} Y(x)) \to Y(x)$ defined by $\mathrm{ev}_x(f) :\equiv f(x)$. Then one could define $\mathrm{ptw}_{f,g}(p, x) :\equiv \mathrm{ap}_{\mathrm{ev}_x}(p)$. The functions provided by these two definitions are not equal by definition, but they can be identified, and one can easily be used in place of the other.

Conversely, given $f, g : \prod_{x:X} Y(x)$, from a basic axiom called *function extensionality*, postulated below in Principle 2.9.17, an identity $f \overset{=}{\to} g$ can be produced from a family of identities of type $f(x) \overset{=}{\to} g(x)$ parametrized by a variable x of type X.

def:naturality-square-commutativity

Definition 2.6.5. Let X, Y be types and $f, g : X \to Y$ functions. Given an element h of type $\prod_{x:X} f(x) \overset{=}{\to} g(x)$, elements x and x' of X, and a path

$p : x \xrightarrow{=} x'$, we have two elements $h(x') \cdot \mathrm{ap}_f(p)$ and $\mathrm{ap}_g(p) \cdot h(x)$ of type $f(x) \xrightarrow{=} g(x')$. We construct an identification

$$\mathrm{ns}(h, p) : h(x') \cdot \mathrm{ap}_f(p) \xrightarrow{=} \mathrm{ap}_g(p) \cdot h(x),$$

between them by induction, by setting $\mathrm{ns}(h, \mathrm{refl}_x)$ to be some previously constructed element of $h(x) \cdot \mathrm{refl}_{f(x)} \xrightarrow{=} h(x)$. The type of $\mathrm{ns}(h, p)$ can be depicted as a square[25] and $\mathrm{ns}(h, p)$ is called a *naturality square*. ⌟

[25]
$$\begin{tikzcd}
f(x) \arrow[r, "\mathrm{ap}_f(p)", "="'] \arrow[d, "h(x)"', "="] & f(x') \arrow[d, "h(x')", "="'] \\
g(x) \arrow[r, "="', "\mathrm{ap}_g(p)"'] & g(x')
\end{tikzcd}$$

2.7 *Identifying elements in members of families of types*

If $Y(x)$ is a family of types parametrized by a variable x of type X, and a and a' are elements of type X, then after identifying a with a' it turns out that it is possible to "identify" an element of $Y(a)$ with an element of $Y(a')$, in a certain sense. That is the idea of the following definition.

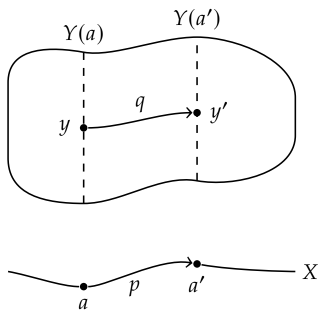
def:pathsoverpaths

DEFINITION 2.7.1. Suppose we are given a type X in a universe $\mathcal{U}$ and a family of types $Y(x)$, also in $\mathcal{U}$, parametrized by a variable x of type X. Given elements $a, a' : X$, $y : Y(a)$, and $y' : Y(a')$ and a path $p : a \xrightarrow{=} a'$, we define a new type $y \underset{p}{\xrightarrow{=}} y'$ in $\mathcal{U}$ as follows. We proceed by induction on a' and p, which reduces us to the case where a' is a and p is refl_a, rendering y and y' of the same type $Y(a)$ in $\mathcal{U}$, allowing us to define $y \xrightarrow[\mathrm{refl}_a]{=} y'$ to be $y \xrightarrow{=} y'$, which is also in $\mathcal{U}$. ⌟

An element $q : y \underset{p}{\xrightarrow{=}} y'$ is called an *identification* of y with y' *over* p, or a *path* from y to y' *over* p. Intuitively, we regard p as specifying a way for a to change gradually into a', and this provides a way for $Y(a)$ to change gradually into $Y(a')$; then q charts a way for y to change gradually into y' as $Y(a)$ changes gradually into $Y(a')$.[26]

ft:path-over-pic

[26]We picture this as follows: the path from y to y' over p travels through the vertical lines representing the types $Y(x)$ as $x : X$ moves along the path p in X from a to a':



The following definition identifies the type of paths over p with a type of paths using transport along p.

def:pathover-trp

DEFINITION 2.7.2. In the context of Definition 2.7.1, define by induction on p an identification $\mathrm{po}_p : \left(y \underset{p}{\xrightarrow{=}} y'\right) \xrightarrow{=} \left(\mathrm{trp}^Y_p(y) \xrightarrow{=} y'\right)$ in $\mathcal{U}$, by setting $\mathrm{po}_{\mathrm{refl}_x} :\equiv \mathrm{refl}_{y \xrightarrow{=} y'}$. ⌟

Many of the operations on paths have counterparts for paths over paths. For example, we may define composition of paths over paths as follows.

def:pathovercomposition

DEFINITION 2.7.3. Suppose we are given a type X and a family of types $Y(x)$ parametrized by the elements x of X. Suppose also that we have elements $x, x', x'' : X$, a path $p : x \xrightarrow{=} x'$, and a path $p' : x' \xrightarrow{=} x''$. Suppose further that we have elements $y : Y(x)$, $y' : Y(x')$, and $y'' : Y(x'')$, with paths $q : y \underset{p}{\xrightarrow{=}} y'$ over p and $q' : y' \underset{p'}{\xrightarrow{=}} y''$ over p'. Then we define the *composite* path $q' \mathbin{\underline{\circ}} q : y \xrightarrow[p' \circ p]{=} y''$ over $p' \circ p$ as follows. First we apply path induction on x'' and p' to reduce to the case where x'' is x' and p' is $\mathrm{refl}_{x'}$. That also reduces the type $y' \underset{p'}{\xrightarrow{=}} y''$ to the identity type $y' \xrightarrow{=} y''$, so we may apply path induction on y'' and q' to reduce to the case where y'' is y' and q' is $\mathrm{refl}_{y'}$. Now observe that $p' \circ p$ is p, so q provides the element we need. ⌟

Similarly, one can define the inverse of a path over a path, writing $q^{-1} : b' \xrightarrow[p^{-1}]{=} b$ for the inverse of $q : b \underset{p}{\xrightarrow{=}} b'$. These operations on paths over paths satisfy many of the laws satisfied by the corresponding operations on paths, after some modification. We will state these laws when we need them.[27]

[27]Exercise: Try to state some of these laws yourself.

The following construction shows how to handle application of a dependent function f to paths using the definition above.

def:apd

Definition 2.7.4. Suppose we are given a type X, a family of types $Y(x)$ parametrized by the elements x of X, and a function $f : \prod_x Y(x)$. Given elements $x, x' : X$ and a path $p : x \xrightarrow{=} x'$, we define

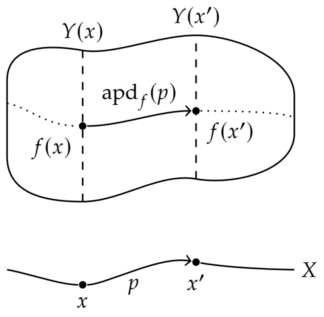
$$\mathrm{apd}_f(p) : f(x) \xrightarrow[p]{=} f(x')$$

by induction on p, setting

$$\mathrm{apd}_f(\mathrm{refl}_x) :\equiv \mathrm{refl}_{f(x)}. \quad \lrcorner$$

The function apd_f is called *dependent application* of f to paths.[28] For convenience, we may abbreviate $\mathrm{apd}_f(p)$ to $f(p)$, when there is no risk of confusion.

[28]We picture f via its *graph* of the values $f(x)$ as x varies in X. The dependent application of f to p is then the piece of the graph that lies over p:



The following construction shows how functions of two variables may be applied to paths over paths.

def:ap1fun2

Definition 2.7.5. Suppose we are given a type X, a family of types $Y(x)$ parametrized by the elements x of X, and a type Z. Suppose also we are given a function $g : \prod_{x:X}(Y(x) \to Z)$ of two variables. Given elements $x, x' : X$, $y : Y(x)$, and $y' : Y(x')$, a path $p : x \xrightarrow{=} x'$, and a path $q : y \xrightarrow[p]{=} y'$ over p, we may construct a path

$$\mathrm{apap}_g(p)(q) : g(x)(y) \xrightarrow{=} g(x')(y')$$

by induction on p and q, setting

$$\mathrm{apap}_g(\mathrm{refl}_x)(\mathrm{refl}_y) :\equiv \mathrm{refl}_{g(x)(y)}. \quad \lrcorner$$

The function $p \mapsto q \mapsto \mathrm{apap}_g(p)(q)$ is called *application* of g to paths over paths. For convenience, we may abbreviate $\mathrm{apap}_g(p)(q)$ to $g(p)(q)$.

The following simple lemma will be useful later.

def:ap1fun2comp

Definition 2.7.6. Suppose we are given a type X, a family of types $Y(x)$ parametrized by the elements x of X, and a type Z. Suppose also we are given a function $g : \prod_{x:X}(Y(x) \to Z)$ of two variables. Given an element $x : X$, elements $y, y' : Y(x)$, and an identity $q : y \xrightarrow{=} y'$, then we define an identification of type $\mathrm{apap}_g(\mathrm{refl}_x)(q) \xrightarrow{=} \mathrm{ap}_{g(x)}(q)$, by induction on q, thereby reducing to the case where y' is y and q is refl_y, rendering the two sides of the equation equal, by definition, to $\mathrm{refl}_{g(x)(y)}$. $\lrcorner$

sec:sum-types

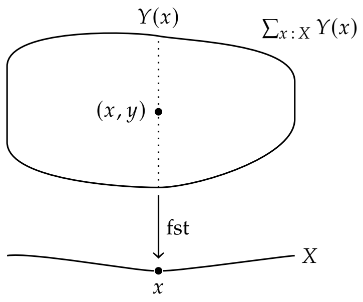
2.8 *Sum types*

There are *sums* of types. By this we mean if X is a type and $Y(x)$ is a family of types parametrized by a variable x of type X, then there will be a type[29] $\sum_{x:X} Y(x)$ whose elements are all pairs (a, b), where $a : X$ and $b : Y(a)$. Since the type of b may depend on a we also call such a pair a *dependent* pair. We may refer to X as the *parameter type* of the sum.[30]

If X and $Y(x)$ are elements of a universe $\mathcal{U}$, then so is $\sum_{x:X} Y(x)$.

[29]Also known as a *Sigma-type*.

[30]We also call $\sum_{x:X} Y(x)$ the *total type* of the family, and we picture it, in the style of the pictures above, as the entire blob lying over X. (Each $Y(x)$ is a vertical line over $x : X$, and a point $y : Y(x)$ becomes a point (x, y) in the blob.)



Proving something about (or constructing something from) every element of $\sum_{x:X} Y(x)$ is done by performing the construction on elements of the form (a, b), for every $a : X$ and $b : Y(a)$. Two important examples of such constructions are:

(1) *first projection*, $\mathrm{fst} : (\sum_{x:X} Y(x)) \to X$, $\mathrm{fst}(a, b) :\equiv a$;

(2) *second projection*, $\mathrm{snd}(a, b) : Y(a)$, $\mathrm{snd}(a, b) :\equiv b$.

it:second-projection

In (2), the type of snd is, in full, $\prod_{z : \sum_{x:X} Y(x)} Y(\mathrm{fst}(z))$.

iterated-sums

Remark 2.8.1. One may consider sums of sums. For example, suppose X is a type, suppose $Y(x)$ is a family of types parametrized by a variable x of type X, and suppose $Z(x, y)$ is a family of types parametrized by variables $x : X$ and $y : Y(x)$. In this case, the *iterated sum* $\sum_{x:X} \sum_{y:Y(x)} Z(x, y)$ consists of pairs of the form $(x, (y, z))$. For simplicity, we introduce the notation $(x, y, z) :\equiv (x, (y, z))$, and refer to (x, y, z) as a *triple* or as a 3-*tuple*.

That process can be repeated: suppose X_1 is a type, suppose $X_2(x_1)$ is a family of types parametrized by a variable x_1 of type X_1, suppose $X_3(x_1, x_2)$ is a family of types parametrized by variables $x_1 : X_1$ and $x_2 : X_2(x_1)$, and so on, up to a family $X_n(x_1, \ldots, x_{n-1})$ of types. In this case, the *iterated sum*

$$\sum_{x_1 : X_1} \sum_{x_2 : X_2(x_1)} \cdots \sum_{x_{n-1} : X_{n-1}(x_1, \ldots, x_{n-2})} X_n(x_1, \ldots, x_{n-1})$$

consists of elements of the form $(x_1, (x_2, (\ldots (x_{n-1}, x_n) \ldots)))$; each such element is a pair whose second member is a pair, and so on, so we may refer to it as an *iterated pair*. For simplicity, we introduce the notation $(x_1, x_2, \ldots, x_n)$ for such an iterated pair, and refer to it as an *n-tuple*. ⌟

2.9 *Equivalences*

sec:equivalence

Using a combination of sum, product, and identity types allows us to express important notions, as done in the following definitions.

The property that a type X has "exactly one element" may be made precise by saying that X has an element such that every other element is equal to it. This property is encoded in the following definition.

def:contractible

Definition 2.9.1. Given a type X, define a type $\mathrm{isContr}(X)$ by setting

$$\mathrm{isContr}(X) :\equiv \sum_{c:X} \prod_{x:X} (c \overset{=}{\to} x). \quad ⌟$$

If $(c, h) : \mathrm{isContr}(X)$, then c will be called the *center* of the the *contraction* h, and we call the type X *contractible*.

By path composition, one sees that any element $x : X$ can serve as the center of a contraction of a contractible type X.

The following lemma gives an important example of a contractible type.

Give a type X and an element a of X, the *singleton type* $\sum_{x:X}(a \overset{=}{\to} x)$ consists of pairs (x, i) with $i : a \overset{=}{\to} x$. The following lemma shows that a singleton type has exactly one element, justifying the name.

lem:thepathspaceiscontractible

Lemma 2.9.2. *For any type X and $a : X$, the singleton type $\sum_{x:X}(a \overset{=}{\to} x)$ is contractible.*

Proof. Take as center the pair (a, refl_a). We have to produce, for any element x of X and for any identification $i : a \xrightarrow{=} x$, an identification of type $(a, \mathrm{refl}_a) \xrightarrow{=} (x, i)$. This is done by path induction on x and i, which reduces us to producing an identity of type $(a, \mathrm{refl}_a) \xrightarrow{=} (a, \mathrm{refl}_a)$; reflexivity provides one, namely $\mathrm{refl}_{(a,\mathrm{refl}_a)}$. □

def:fiber

Definition 2.9.3. Given a function $f : X \to Y$ and an element $y : Y$, the *fiber* (or *preimage*) $f^{-1}(y)$ is encoded by defining

$$f^{-1}(y) :\equiv \sum_{x:X} (y \xrightarrow{=} f(x)).$$

In other words, an element of the fiber $f^{-1}(y)$ is a pair consisting of an element x of X and an identification of type $y \xrightarrow{=} f(x)$. ⌟

In set theory, a function $f : X \to Y$ is a bijection if and only if all preimages $f^{-1}(y)$ consist of exactly one element. We can also express this in type theory, in a definition due to Voevodsky, for types in general.

def:equivalence

Definition 2.9.4. A function $f : X \to Y$ is called an *equivalence* if $f^{-1}(y)$ is contractible for all $y : Y$. The condition is encoded by the type

$$\mathrm{isEquiv}(f) :\equiv \prod_{y:Y} \mathrm{isContr}(f^{-1}(y)). \quad ⌟$$

We may say that X and Y are *equivalent* if there is an equivalence between them.

def:type-of-equivalences

Definition 2.9.5. We define the type $X \xrightarrow{\simeq} Y$ of equivalences from X to Y by the following definition.

$$(X \xrightarrow{\simeq} Y) :\equiv \sum_{f:X\to Y} \mathrm{isEquiv}(f). \quad ⌟$$

Suppose $f : X \xrightarrow{\simeq} Y$ is an equivalence, and let $t(y) : \mathrm{isContr}(f^{-1}(y))$, for each $y : Y$, be the corresponding witness to contractibility of the fiber. Using t we can define an inverse function $g : Y \to X$ by setting $g(y) :\equiv \mathrm{fst}(\mathrm{fst}(t(y)))$.[31]

[31] Note that $\mathrm{fst}(t(y)) : f^{-1}(y)$, so $\mathrm{fst}(\mathrm{fst}(t(y))) : X$ with $\mathrm{snd}(\mathrm{fst}(t(y))) : y \xrightarrow{=} f(\mathrm{fst}(\mathrm{fst}(t(y))))$.

There is an identification of type $f(g(y)) \xrightarrow{=} y$, which can be seen by unfolding all the definitions. Moreover, we have $(x, \mathrm{refl}_{f(x)}) : f^{-1}(f(x))$, with the latter as the fiber that $t(f(x))$ proves contractible. Hence the center of contraction $\mathrm{fst}(t(f(x))$ is equal to $(x, \mathrm{refl}_{f(x)})$, and so $g(f(x)) \equiv (\mathrm{fst}(\mathrm{fst}(t(f(x))) \xrightarrow{=} x$.

We have shown that f and g are inverse functions. When it won't cause confusion with the notation for the fibers of f, we will write f^{-1} instead of g.

For any type X, the identity function id_X is an equivalence from X to X. To see that, observe that for every element a in X, $\mathrm{id}_X^{-1}(a)$ is a singleton type and hence is contractible. This observation, combined with the fact that $\mathrm{trp}^T_{\mathrm{refl}_x} \equiv \mathrm{id}_{T(x)}$, gives that the function trp^T_e from Definition 2.5.4 is an equivalence from $T(x)$ to $T(y)$, for all $e : x \xrightarrow{=} y$.

xca:equivalence-invers

Exercise 2.9.6. Make sure you understand the two applications of fst in the definition $f^{-1}(y) :\equiv \mathrm{fst}(\mathrm{fst}(t(y)))$ above. Show that f^{-1} is an equivalence from Y to X. Give a function $(X \simeq Y) \to (Y \simeq X)$. ⌟

xca:equivalence-comp

Exercise 2.9.7. Give a function $(X \simeq Y) \to ((Y \simeq Z) \to (X \simeq Z))$. ⌟

xca:2-out-of-3

Exercise 2.9.8. Consider types A, B, and C, functions $f : A \to B$, $g : A \to C$ and $h : B \to C$, together with an element $e : hf \overset{=}{\to} g$. Prove that if two of the three functions are equivalences, then so is the third one. ⌟

The following lemma gives an equivalent characterization of equivalence that is sometimes easy to use.

lem:weq-iso

Lemma 2.9.9. *Let X, Y be types. For each equivalence $f : X \to Y$, we have a function $g : Y \to X$ such that for all $x : X$ we have $g(f(x)) \overset{=}{\to} x$ and for all $y : Y$ we have $f(g(y)) \overset{=}{\to} y$. Conversely, if we have such a function g, then f is an equivalence.*

Proof. Since $f : X \to Y$ is an equivalence we can take $g :\equiv f^{-1}$. For the converse, see Chapter 4 of the HoTT Book,[32] or `isweq_iso`. □

[32] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: `https://homotopytypetheory.org/book`, 2013.

We put Lemma 2.9.9 immediately to good use.

lem:contract-away

Lemma 2.9.10. *Let X be a type with element a, and let $B(x, i)$ be a type for all $x : X$ and $i : a \overset{=}{\to} x$. Define $f(x, i) : B(x, i) \to B(a, \mathrm{refl}_a)$ by induction on i, setting $f(a, \mathrm{refl}_a, b) :\equiv b$ for all $b : B(a, \mathrm{refl}_a)$. Then f defines an equivalence*

$$f \;:\; \sum_{x : X} \sum_{i : a \overset{=}{\to} x} B(x, i) \quad\to\quad B(a, \mathrm{refl}_a).$$

Proof. We can also define $g : B(a, \mathrm{refl}_a) \to \sum_{x : X} \sum_{i : a \overset{=}{\to} x} B(x, i)$ mapping $b : B(a, \mathrm{refl}_a)$ to (a, refl_a, b). Clearly $f(g(b)) \overset{=}{\to} b$ for all $b : B(a, \mathrm{refl}_a)$. Moreover, $g(f(x, i, b)) \overset{=}{\to} (x, i, b)$ is clear by induction on i, for all $b : B(x, i)$. By Lemma 2.9.9 it follows that f is an equivalence. □

The above lemma clearly reflects the contractibility of the singleton type $\sum_{x : X}(a \overset{=}{\to} x)$.[33] For this reason we call application of this lemma 'to contract away' the prefix $\sum_{x : X} \sum_{i : a \overset{=}{\to} x}$, in order to obtain a simpler type. It is often applied in the following simpler form.

[33] In fact, an alternative proof would go as follows: First, we use Lemma 2.9.9 to show associativity of sum types, i.e., $\sum_{x : X} \sum_{y : Y(x)} Z(x, y) \simeq \sum_{w : (\sum_{x : X} Y(x))} Z(\mathrm{fst}\, w, \mathrm{snd}\, w)$, where X is a type, $Y(x)$ is a family of types depending on $x : X$, and $Z(x, y)$ is a family of types depending on $x : X$ and $y : Y(x)$. Then, we show for any contractible type X and for any family of types $Y(x)$ depending on $x : X$, that there is an equivalence between $\sum_{x : X} Y(x)$ and $Y(c)$, where c is the center of contraction.

cor:contract-away

Corollary 2.9.11. *With conditions as above, but with B not depending on i, the same f establishes an equivalence*

$$\sum_{x : X}((a \overset{=}{\to} x) \times B(x)) \quad\simeq\quad B(a).$$

In the direction of further generality, we offer the following exercise.

xca:sum-equiv-base

Exercise 2.9.12. Suppose X, Y are types related by an equivalence $f : X \to Y$. Let $B(x)$ be a type for all $x : X$. Construct an equivalence between $\sum_{x : X} B(x)$ and $\sum_{y : Y} B(f^{-1}(y))$. ⌟

We proceed now to define the notion of fiberwise equivalence.

def:fiberwise

Definition 2.9.13. Let X be a type, and let $Y(x), Z(x)$ be families of types parameterised by $x : X$. A map f of type $\prod_{x : X}(Y(x) \to Z(x))$ can be viewed as a family of maps $f(x) : Y(x) \to Z(x)$ and is called a *fiberwise* map. The *totalization* of f is defined as

$$\mathrm{tot}(f) : \Big(\sum_{x : X} Y(x)\Big) \to \sum_{x : X} Z(x),$$

setting $\mathrm{tot}(f)(x, y) :\equiv (x, f(x)(y))$. ⌟

lem:fiberwise

Lemma 2.9.14. *Let conditions be as in Definition 2.9.13. If $f(x) : Y(x) \to Z(x)$ is an equivalence for every $x : X$ (we say that f is a* fiberwise *equivalence), then $\mathrm{tot}(f)$ is an equivalence.*

We will allow ourselves to drop the "fiberwise" and talk simply about maps and equivalences between type families.

Proof. If $f(x) : Y(x) \to Z(x)$ is an equivalence for all x in X, then the same is true of all $f(x)^{-1} : Z(x) \to Y(x)$. Then we have the totalization $\mathrm{tot}(x \mapsto f(x)^{-1})$, which can easily be proved to be an inverse of $\mathrm{tot}(f)$ (see the next exercise). Now apply Lemma 2.9.9. □

xca:fiberwise

EXERCISE 2.9.15. Complete the details of the proof of Lemma 2.9.14. ⌟

The converse to Lemma 2.9.14 also holds.

lem:fiberwise-equiv-from-tot

LEMMA 2.9.16. *Continuing with the setup of Definition 2.9.13, if* $\mathrm{tot}(f)$ *is an equivalence, then* f *is a fiberwise equivalence.*

For a proof see Theorem 4.7.7 of the HoTT Book[34].

[34] Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*.

Yet another application of the notion of equivalence is to postulate axioms.

def:funext

PRINCIPLE 2.9.17. The axiom of *function extensionality* postulates that the function $\mathrm{ptw}_{f,g} : f \xrightarrow{=} g \to \prod_{x:X} f(x) \xrightarrow{=} g(x)$ in Definition 2.6.4 is an equivalence. Formally, we postulate the existence of an element $\mathrm{funext} : \mathrm{isEquiv}(\mathrm{ptw}_{f,g})$. From that we can construct the corresponding inverse function.

$$\mathrm{ptw}_{f,g}^{-1} : \Big(\prod_{x:X} f(x) \xrightarrow{=} g(x)\Big) \to f \xrightarrow{=} g.$$

Thus two functions whose values can all be identified can themselves be identified. This supports the intuition that there is nothing more to a function than the values it sends its arguments to. ⌟

EXERCISE 2.9.18. Let X be a type. Construct an equivalence of type $(\mathrm{True} \to X) \xrightarrow{\simeq} X$. ⌟

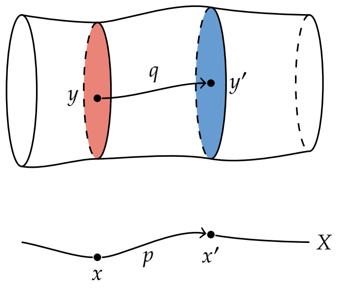
EXERCISE 2.9.19. Let X be a type, and regard True as a constant family of types over X. Construct an equivalence of type $(\sum_{x:X} \mathrm{True}) \xrightarrow{\simeq} X$. ⌟

2.10 *Identifying pairs*

sec:pairpaths

Equality of two elements of $\sum_{x:X} Y(x)$ is inductively defined in Section 2.5, as for any other type, but one would like to express identity between pairs in terms of identifications in the constituent types. This would explain better what it means for two pairs to be identified. We start with a definition.

We picture paths between pairs much in the same way as paths over paths, cf. Footnote 26. Just as, to give a pair in the sum type $\sum_{x:X} Y(x)$, we need both the point x in the parameter type X as well as the point y in $Y(x)$, to give a path from (x,y) to (x',y'), we need both a path $p : x \xrightarrow{=} x'$ as well as a path $q : y \xrightarrow[p]{=} y'$ over p. Here's a similar picture, where we depict the types in the family as being 2-dimensional for a change.



def:pairtopath

DEFINITION 2.10.1. Suppose we are given a type X and a family of types $Y(x)$ parametrized by the elements x of X. Consider the function

$$\mathrm{pair} : \prod_{x:X} \Big(Y(x) \to \sum_{x':X} Y(x')\Big)$$

defined by

$$\mathrm{pair}(x)(y) :\equiv (x,y).$$

For any elements (x,y) and (x',y') of $\sum_{x:X} Y(x)$, we define the map

$$\Big(\sum_{p : x \xrightarrow{=} x'} y \xrightarrow[p]{=} y'\Big) \to \big((x,y) \xrightarrow{=} (x',y')\big)$$

by

$$(p,q) \mapsto \mathrm{apap}_{\mathrm{pair}}(p)(q).$$

(Refer to Definition 2.7.1 for the meaning of the type $y \xrightarrow[p]{=} y'$, and to Definition 2.7.5 for the definition of apap.) We introduce $\overline{(p,q)}$ as notation for $\mathrm{apap}_{\mathrm{pair}}(p)(q)$. ⌟

cor:isEq-pair=

Lemma 2.10.2. *In the situation of Definition 2.10.1, if x' is x, so that we have $(y \xrightarrow[\mathrm{refl}_x]{=} y') \equiv (y \xrightarrow{=} y')$, then for any $q : y \xrightarrow{=} y'$, we can construct an identification of type*

$$\overline{(\mathrm{refl}_x, q)} \xrightarrow{=} \mathrm{ap}_{\mathrm{pair}(x)}\, q$$

holds.

Proof. By induction on q it suffices to establish the identity

$$\overline{(\mathrm{refl}_x, \mathrm{refl}_y)} \xrightarrow{=} \mathrm{ap}_{\mathrm{pair}(x)}(\mathrm{refl}_y),$$

both sides of which are equal to $\mathrm{refl}_{(x,y)}$ by definition. □

The following lemma gives the desired characterization of paths between pairs.

lem:isEq-pair=

Lemma 2.10.3. *Suppose we are given a type X and a family of types $Y(x)$ parametrized by the elements x of X. For any elements (x,y) and (x',y') of $\sum_{x:X} Y(x)$, the map defined in Definition 2.10.1 defined by*

$$(p,q) \mapsto \overline{(p,q)}$$

is an equivalence of type

$$\left(\sum_{p : x \xrightarrow{=} x'} y \xrightarrow[p]{=} y' \right) \simeq \left((x,y) \xrightarrow{=} (x',y') \right).$$

Proof. Call the map Φ. A map the other way,

$$\Psi : ((x,y) \xrightarrow{=} (x',y')) \to \sum_{p : x \xrightarrow{=} x'} y \xrightarrow[p]{=} y',$$

can be defined by induction, by setting

$$\Psi(\mathrm{refl}_{(x,y)}) :\equiv (\mathrm{refl}_x, \mathrm{refl}_y).$$

One proves, by induction on paths, the identities $\Psi(\Phi(p,q)) \xrightarrow{=} (p,q)$ and $\Phi(\Psi(r)) \xrightarrow{=} r$, so Ψ and Φ are inverse functions. Applying Lemma 2.9.9, we see that Φ and Ψ are inverse equivalences, thereby obtaining the desired result. □

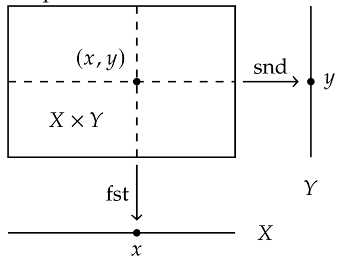
We often use $\mathrm{fst}(\overline{(p,q)}) \xrightarrow{=} p$ and $\mathrm{snd}(\overline{(p,q)}) \xrightarrow{=} q$, which follow by induction on p and q from the definitions of ap and $\overline{(\_,\_)}$. Similarly, $r \xrightarrow{=} \overline{(\mathrm{fst}(r), \mathrm{snd}(r))}$ by induction on r.

sec:binprod-types

2.11 *Binary products*

There is special case of sum types that deserves to be mentioned since it occurs quite often. Let X and Y be types, and consider the constant family of types $Y(x) :\equiv Y$. In other words, $Y(x)$ is a type that depends on an element x of X that happens to be Y for any such x. (Recall Exercise 2.5.6.) Then we can form the sum type $\sum_{x:X} Y(x)$ as above. Elements of this sum type are pairs (x,y) with x in X and y in $Y(x) \equiv Y$.[35]

[35]These *cartesian* products we illustrate as usual by rectangles where one side represents X and the other Y.



In this case the type of y doesn't depend on x, and in this special case the sum type is called the *binary product*, or *cartesian product* of the types X and Y, denoted by $X \times Y$.

At first glance, it might seem odd that a sum is also a product, but exactly the same thing happens with numbers, for the sum $5+5+5$ is also referred to as the product 3×5. Indeed, that's one way to define 3×5.

Recall that we have seen something similar with the product type $\prod_{x:X} Y(x)$, which we let $X \to Z$ denote in the case where $Y(x)$ is a constant family of the form $Y(x) :\equiv Z$, for some type Z.

The type $X \times Y$ inherits the functions fst, snd from $\sum_{x:X} Y(x)$, with the same definitions $\mathrm{fst}(x,y) :\equiv x$ and $\mathrm{snd}(x,y) :\equiv y$. Their types can now be denoted in a simpler way as $\mathrm{fst} : (X \times Y) \to X$ and $\mathrm{snd} : (X \times Y) \to Y$, and they are called as before the first and the second projection, respectively.

Again, proving something about (or constructing something from) every element (a,b) of $X \times Y$ is simply done for all $a : X$ and $b : Y$.

There is an equivalence between $(a_1,b_1) \xrightarrow{=} (a_2,b_2)$ and $(a_1 \xrightarrow{=} a_2) \times (b_1 \xrightarrow{=} b_2)$. This follows from Lemma 2.10.3 together with Exercise 2.5.6.

If $f : X \to Y$ and $f' : X' \to Y'$, then we let $f \times f'$ denote the map of type $(X \times X') \to (Y \times Y')$ that sends (x,x') to $(f(x), f'(x'))$.

The following lemma follows from Lemma 2.10.3, combined with Definition 2.7.2 and Exercise 2.5.6.

lem:isEq-pair-bin=

LEMMA 2.11.1. *Suppose we are given type X and Y. For any elements (x,y) and (x',y') of $X \times Y$, the map defined in Definition 2.10.1 defined by*

$$(p,q) \mapsto \overline{(p,q)}$$

is an equivalence of type

$$(x \xrightarrow{=} x') \times (y \xrightarrow{=} y') \simeq \big((x,y) \xrightarrow{=} (x',y')\big)\,.$$

xca:binary-prod-equiv

EXERCISE 2.11.2. Let X, Y be types in a universe $\mathcal{U}$, and consider the type family $T(z)$ in $\mathcal{U}$ depending on $z : \mathrm{Bool}$ defined by $T(\mathrm{no}) :\equiv X$ and $T(\mathrm{yes}) :\equiv Y$. Show that the function $(\prod_{b:\mathrm{Bool}} T(b)) \to X \times Y$ sending f to $(f(\mathrm{no}), f(\mathrm{yes}))$, is an equivalence. ⌟

EXERCISE 2.11.3. Let X be a type. Construct an equivalence of type $(X \times \mathrm{True}) \xrightarrow{\simeq} X$. ⌟

2.12 *More inductive types*

sec:inductive-types

There are other examples of types that are conveniently introduced in the same way as we have seen with the natural numbers and the identity types. A type presented in this style shares some common features: there are some ways to create new elements, and there is a way (called *induction*) to prove something about every element of the type (or family of types). We will refer to such types as *inductive* types, and we present a few more of them in this section, including the finite types, and then we present some other constructions for making new types from old ones. For each of these constructions we explain what it means for two elements of the newly constructed type to be equal in terms of identity in the constituent types.

2.12.1 *Finite types*

sec:finite-types

Firstly, there is the *empty* type in the universe $\mathcal{U}_0$, denoted by $\emptyset$ or by False. It is an inductive type, with no way to construct elements of it. The induction principle for $\emptyset$ says that to prove something about (or to construct something from) every element of $\emptyset$, it suffices to consider no special cases (!). Hence, every statement about an arbitrary element of $\emptyset$ can be proven. (This logical principle is traditionally called *ex falso quodlibet*.[36]) As an example, we may prove that any two elements x and y of $\emptyset$ are equal by using induction on x.

[36]From falsehood, anything follows.

An element of $\emptyset$ will be called an *absurdity*. Of course, one expects that there are no real absurdities in mathematics, nor in any logical system (such as ours) that attempts to provide a language for mathematics, but it is important to have such a name so we can discuss the possibility, which might result inadvertently from the introduction of unwarranted assumptions. For example, to assert that a type P has no elements, it would be sensible to assert that an element of P would lead to an absurdity. Providing a function of type $P \to \emptyset$ is a convenient way to make that assertion. Hence we define the *negation* of a type by setting $\neg P :\equiv (P \to \emptyset)$. Using it, we may define the type $(a \neq b) :\equiv \neg(a \xrightarrow{=} b)$; an element of it asserts that a and b cannot be identified.

Secondly, there will also be an inductive type called True in the universe $\mathcal{U}_0$ provided with a single element triv; (the name triv comes from the word "trivial"). Its induction principle states that, in order to prove something about (or to construct something from) every element of True, it suffices to consider the special case where the element is triv. As an example, we may construct, for any element $u : \mathrm{True}$, an identity of type $u \xrightarrow{=} \mathrm{triv}$; we use induction to reduce to the case where u is triv, and then $\mathrm{refl}_{\mathrm{triv}}$ provides the desired element. One may also construct, for any elements x and y of True, an identity of type $x \xrightarrow{=} y$ by using induction both on x and on y.

There is a function $X \to \mathrm{True}$, for any type X, namely: $a \mapsto \mathrm{triv}$. This corresponds, for propositions, to the statement that an implication holds if the conclusion is true.

xca:True-univ-prop

Exercise 2.12.2. Let X be a type. Define the function e of type $(\mathrm{True} \to X) \to X$ by $e(f) :\equiv f(\mathrm{triv})$. Prove that e is an equivalence. This is called *the universal property of* True. ⌟

Thirdly, there will be an inductive type called Bool in the universe $\mathcal{U}_0$, provided with two elements, yes and no. Its induction principle states that, in order to prove something about (or to construct something from) every element of Bool, it suffices to consider two cases: the special case where the element is yes and the special case where the element is no.

We may use substitution to construct an element of type yes $\neq$ no. To do this, we introduce a family of types $P(b)$ in the universe $\mathcal{U}_0$ parametrized by a variable $b : \mathrm{Bool}$. We define $P(b)$ by induction on b by setting $P(\mathrm{yes}) :\equiv \mathrm{True}$ and $P(\mathrm{no}) :\equiv \mathrm{False}$. (The definition of $P(b)$ is motivated by the expectation that we will be able to construct an equivalence between $P(b)$ and $b \xrightarrow{=} \mathrm{yes}$.) If there were an element $e : \mathrm{yes} \xrightarrow{=} \mathrm{no}$, we could substitute no for yes in $\mathrm{triv} : P(\mathrm{yes})$ to get an element of $P(\mathrm{no})$, which is absurd. Since e was arbitrary, we have defined a function $(\mathrm{yes} \xrightarrow{=} \mathrm{no}) \to \emptyset$, as desired.

In the same way, we may use substitution to prove that successors of natural numbers are never equal to 0, i.e., for any $n:\mathbb{N}$ that $0 \neq \mathrm{succ}(n)$. To do this, we introduce a family of types $P(i)$ in $\mathcal{U}_0$ parametrized by a variable $i:\mathbb{N}$. Define P recursively by specifying that $P(0) :\equiv \mathrm{True}$ and $P(\mathrm{succ}(m)) :\equiv \mathrm{False}$. (The definition of $P(i)$ is motivated by the expectation that we will be able to construct an equivalence between $P(i)$ and $i \xrightarrow{=} 0$.) If there were an element $e : 0 \xrightarrow{=} \mathrm{succ}(n)$, we could substitute $\mathrm{succ}(n)$ for 0 in $\mathrm{triv} : P(0)$ to get an element of $P(\mathrm{succ}(n))$, which is absurd. Since e was arbitrary, we have defined a function $(0 \xrightarrow{=} \mathrm{succ}(n)) \to \emptyset$, establishing the claim.

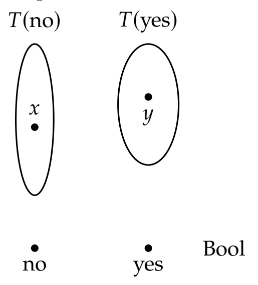
In a similar way we will in Section 2.24 define types $\mathbb{n}$ for any n in $\mathbb{N}$ such that $\mathbb{n}$ is a type (set) of n elements.

2.12.3 *Binary sums*

sec:binsum-types

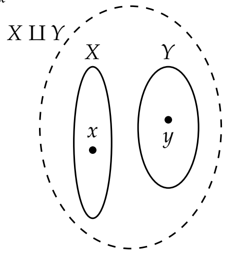
For sum types of the form $\sum_{b:\mathrm{Bool}} T(b)$, with $T(b)$ a type depending on b in Bool, there is an equivalence with a simpler type.[37] After all, the type family $T(b)$ is fully determined by two types, namely by the types $T(\mathrm{no})$ and $T(\mathrm{yes})$. The elements of $\sum_{b:\mathrm{Bool}} T(b)$ are dependent pairs (no, x) with x in $T(\mathrm{no})$ and (yes, y) with y in $T(\mathrm{yes})$. The resulting type can be viewed as the *disjoint union* of $T(\mathrm{no})$ and $T(\mathrm{yes})$: from an element of $T(\mathrm{no})$ or an element of $T(\mathrm{yes})$ we can produce an element of $\sum_{b:\mathrm{Bool}} T(b)$.

[37] In a case like this, we can thicken up the lines denoting $T(\mathrm{no})$ and $T(\mathrm{yes})$ in our picture, if we like:



These disjoint union types can be described more clearly in the following way. The *binary sum* of two types X and Y, denoted $X \amalg Y$, is an inductive type with two constructors: $\mathrm{inl} : X \to X \amalg Y$ and $\mathrm{inr} : Y \to X \amalg Y$.[38] Proving a property of any element of $X \amalg Y$ means proving that this property holds of any inl_x with $x : X$ and any inr_y with $y : Y$. In general, constructing a function f of type $\prod_{z:X\amalg Y} T(z)$, where $T(z)$ is a type depending on z, is done by defining $f(\mathrm{inl}_x)$ for all x in X and $f(\mathrm{inr}_y)$ for all y in Y.

[38] Be aware that in a picture, the same point may refer either to x in X or to inl_x in the sum $X \amalg Y$:



xca:binary-sum-equiv

EXERCISE 2.12.4. Let X, Y be types in a universe $\mathcal{U}$, and consider the type family $T(z)$ in $\mathcal{U}$ depending on $z : \mathrm{Bool}$ defined by induction on z by $T(\mathrm{no}) :\equiv X$ and $T(\mathrm{yes}) :\equiv Y$. Show that the map $f : X \amalg Y \to \sum_{b:\mathrm{Bool}} T(b)$, defined by $f(\mathrm{inl}_x) :\equiv (\mathrm{no}, x)$ and $f(\mathrm{inr}_y) :\equiv (\mathrm{yes}, y)$, is an equivalence. ⌟

Identification of two elements a and b in $X \amalg Y$ is only possible if they are constructed with the same constructor. Thus $\mathrm{inl}_x \xrightarrow{=} \mathrm{inr}_y$ is always empty, and there are equivalences of type $(\mathrm{inl}_x \xrightarrow{=} \mathrm{inl}_{x'}) \simeq (x \xrightarrow{=} x')$ and $(\mathrm{inr}_y \xrightarrow{=} \mathrm{inr}_{y'}) \simeq (y \xrightarrow{=} y')$.

xca:binary-sum-id

EXERCISE 2.12.5. Prove these statements using Exercise 2.12.4, Lemma 2.10.3, and a characterization of the identity types of Bool. ⌟

xca:bin-sum-univ-prop

EXERCISE 2.12.6. Let X, Y, Z be types. Define a function e from $(X \amalg Y) \to Z$ to $(X \to Z) \times (Y \to Z)$ by precomposition with the constructors. Prove that e is an equivalence. This is called *the universal property of the binary sum*. ⌟

EXERCISE 2.12.7. Let X be a type. Construct an equivalence of type $(X \amalg \emptyset) \xrightarrow{\simeq} X$. ⌟

2.12.8 *Unary sums*

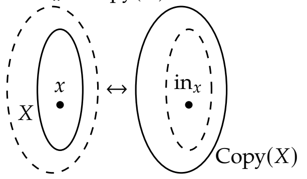
sec:unary-sum-types

Sometimes it is useful to be able to make a copy of a type X: A new type that behaves just like X, though it is not definitionally equal to X. The *unary sum* or *wrapped copy* of X is an inductive type $\mathrm{Copy}(X)$ with a single constructor, $\mathrm{in} : X \to \mathrm{Copy}(X)$.[39] Constructing a function $f : \prod_{z:\mathrm{Copy}(X)} T(z)$, where $T(z)$ is a type depending on $z : \mathrm{Copy}(X)$, is done by defining $f(\mathrm{in}_x)$ for all $x : X$. Taking $T(z)$ to be the constant family at X, we get a function, $\mathrm{out} : \mathrm{Copy}(X) \to X$, called the *destructor*, with $\mathrm{out}(\mathrm{in}_x) :\equiv x$ for $x : X$, and the induction principle implies that $\mathrm{in}_{\mathrm{out}(z)} \overset{=}{\to} z$ for all $z : \mathrm{Copy}(X)$, so there is an equivalence of type $\mathrm{Copy}(X) \simeq X$, as expected. In fact, we will assume that the latter equation even holds definitionally. It follows that there are equivalences of type $(\mathrm{in}_x \overset{=}{\to} \mathrm{in}_{x'}) \simeq (x \overset{=}{\to} x')$ and $(\mathrm{out}(z) \overset{=}{\to} \mathrm{out}(z')) \simeq (z \overset{=}{\to} z')$.

Note that we can make several copies of X that are not definitionally equal to each other, for instance, by picking different names for the constructor. We write $\mathrm{Copy}_{\mathrm{con}}(X)$ for a copy of X whose constructor is

$$\mathrm{con} : X \to \mathrm{Copy}_{\mathrm{con}}(X).$$

[39] A point $x : X$ corresponds to the point $\mathrm{in}_x : \mathrm{Copy}(X)$:



Note that $\mathrm{Copy}(X)$ can alternatively be defined as $\sum_{z:\mathrm{True}} X$.

exa:nnn

Example 2.12.9. Here's an example to illustrate why it can be useful to make such a wrapped type: We introduced the natural numbers $\mathbb{N}$ in Section 2.4. Suppose we want a type consisting of negations of natural numbers, $\{\ldots, -2, -1, 0\}$, perhaps as an intermediate step towards building the set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.[40] Of course, the type $\mathbb{N}$ itself would do, but then we would need to pay extra attention to whether $n : \mathbb{N}$ is supposed to represent n as an integer or its negation. So instead we take the wrapped copy $\mathbb{N}^- :\equiv \mathrm{Copy}_-(\mathbb{N})$, with constructor $- : \mathbb{N} \to \mathbb{N}^-$. There is no harm in also writing $- : \mathbb{N}^- \to \mathbb{N}$ for the destructor. This means that there is an equivalence of type $\mathbb{N}^- \simeq \mathbb{N}$, for the elements of $\mathbb{N}^-$ are of the form $-n$ for $n : \mathbb{N}$. Indeed, $-(-n) \equiv n$ for n an element of either $\mathbb{N}$ or $\mathbb{N}^-$, and there is an equivalence of type $(-n \overset{=}{\to} -n') \simeq (n \overset{=}{\to} n')$. ⌟

[40] We implement this in Definition 3.2.1.

2.13 *Univalence*

sec:univax

The univalence axiom, to be presented in this section, greatly enhances our ability to produce identities between the two types and to use the resulting identities to transport (in the sense of Definition 2.5.4) properties and structure between the types. It asserts that if $\mathcal{U}$ is a universe, and X and Y are types in $\mathcal{U}$, then there is an equivalence between identities between X and Y and equivalences between X and Y.

We now define the function that the univalence axiom postulates to be an equivalence.

def:idtoeq

Definition 2.13.1. For types X and Y in a universe $\mathcal{U}$ and a path $p : X \overset{=}{\to} Y$, we define an equivalence $\mathrm{cast}_{X,Y}(p) : X \simeq Y$ by induction on Y and p, setting $\mathrm{cast}_{X,X}(\mathrm{refl}_X) :\equiv \mathrm{id}_X : X \simeq X$. The result is a function

$$\mathrm{cast}_{X,Y} : (X \overset{=}{\to} Y) \to (X \simeq Y). \quad ⌟$$

In expressions such as $\mathrm{cast}_{X,Y}(p)$ we may abbreviate $\mathrm{cast}_{X,Y}$ to cast if no confusion will result. We may also write $\mathrm{cast}(p)$ more briefly as $\tilde{p}$, which we also use to denote the corresponding function from X to Y.[41]

[41] *Cast* is here used in the sense of "arranging into a suitable form". A more theatrical description would be that an element x of X is cast in the *role* of an element of Y as *directed* by the path $p : X \overset{=}{\to} Y$. But beware that some programming languages use *cast* in a different sense: to take a bit-pattern representing an object of type X and simply *reinterpreting* the same bits as an object of type Y.

Let T be a variable of type $\mathcal{U}$; then we may view T as a family of types parametrized by $\mathcal{U}$, of the sort required for use with transport as defined in Definition 2.5.4. One may construct an identity of type $\mathrm{cast}(p)(x) \xrightarrow{=} \mathrm{trp}_p^T(x)$, for $x : X$, by induction on Y and p. As a corollary, one sees that the function trp_p^T is an equivalence.

We are ready to state the univalence axiom.

def:univalence

Principle 2.13.2 (Univalence Axiom). Voevodsky's *univalence axiom* postulates that $\mathrm{cast}_{X,Y}$ is an equivalence for all $X, Y : \mathcal{U}$. Formally, we postulate the existence of an element

$$\mathrm{ua}_{X,Y} : \mathrm{isEquiv}(\mathrm{cast}_{X,Y}). \quad ⌟$$

For an equivalence $f : X \simeq Y$, we will adopt the notation $\mathrm{ua}(f) : X \xrightarrow{=} Y$ to denote $\mathrm{cast}_{X,Y}^{-1}(f)$, the result of applying the inverse function of $\mathrm{cast}_{X,Y}$ to f, if no confusion will result. Thus there are identities of type $\mathrm{cast}(\mathrm{ua}(f)) \xrightarrow{=} f$ and $\mathrm{ua}(\mathrm{cast}(p)) \xrightarrow{=} p$.

We may also write $\mathrm{ua}(f)$ more briefly as $\bar{f}$. Thus there are identities of type $\bar{\bar{p}} \xrightarrow{=} p$ and $\bar{\bar{f}} \xrightarrow{=} f$. There are also identities of type $\overline{\mathrm{id}_X} \xrightarrow{=} \mathrm{refl}_X$ and $\overline{g\,f} \xrightarrow{=} \bar{g}\,\bar{f}$ if $g : Y \simeq Z$.

xca:C2

Exercise 2.13.3. Prove that $\mathrm{Bool} \xrightarrow{=} \mathrm{Bool}$ has exactly two elements, $\mathrm{refl}_{\mathrm{Bool}}$ and twist (where twist is given by univalence from the equivalence $\mathrm{Bool} \to \mathrm{Bool}$ interchanging the two elements of Bool), and that $\mathrm{twist} \cdot \mathrm{twist} \xrightarrow{=} \mathrm{refl}_{\mathrm{Bool}}$. ⌟

2.14 *Heavy transport*

sec:heavy-transport

In this section we collect useful results on transport in type families that are defined by a type constructor applied to families of types. Typical examples of such 'structured' type families are $Y(x) \to Z(x)$ and $x \xrightarrow{=} x$ parametrized by $x : X$.

def:function-type-families

Definition 2.14.1. Let X be a type, and let $Y(x)$ and $Z(x)$ be families of types parametrized by a variable $x : X$. Define $Y \to Z$ to be the type family with $(Y \to Z)(x) :\equiv Y(x) \to Z(x)$. ⌟

Recall from Definition 2.9.13 that an element $f : \prod_{x:X}(Y \to Z)(x)$ is called a fiberwise map, and f is called a fiberwise equivalence, if $f(x) : Y(x) \to Z(x)$ is an equivalence for all $x : X$.

lem:trp-in-function-type

Construction 2.14.2. *Let X be a type, and let $Y(x)$ and $Z(x)$ be types for every $x : X$. Then we have for every $x, x' : X$, $e : x \xrightarrow{=} x'$, $f : Y(x) \to Z(x)$, and $y' : Y(x')$:*

$$\mathrm{trp}_e^{Y \to Z}(f)(y') \xrightarrow{=} \mathrm{trp}_e^Z\Big(f\big(\mathrm{trp}_{e^{-1}}^Y(y')\big)\Big).$$

Implementation of Construction 2.14.2. By induction on $e : x \xrightarrow{=} x'$. For $e \equiv \mathrm{refl}_x$, we have $e^{-1} \equiv \mathrm{refl}_x$, and all transports are identity functions of appropriate type. □

An important special case of the above lemma is with $\mathcal{U}$ as parameter type and type families $Y :\equiv Z :\equiv \mathrm{id}_{\mathcal{U}}$. Then $Y \to Z$ is $X \to X$ as a type depending on $X : \mathcal{U}$. Now, if $A : \mathcal{U}$ and $e : A \xrightarrow{=} A$ comes by applying the univalence axiom to some equivalence $g : A \to A$, then the above lemma combined with function extensionality yields that for any $f : A \to A$

$$\mathrm{trp}_e^{X \mapsto (X \to X)}(f) \xrightarrow{=} g \circ f \circ g^{-1}.$$

This equation is phrased as 'transport by conjugation'. The following lemma is proved by induction on $e : x \xrightarrow{=} x'$.

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "g"] \\ A \arrow[r, "g\circ f\circ g^{-1}", "\mathrm{trp}_{\mathrm{ua}(g)}(f)"'] & A \end{tikzcd}$$

lem:trp-in-fx=Ygx

Construction 2.14.3. *Let X, Y be types, $f, g : X \to Y$ functions, and let $Z(x) :\equiv (f(x) \xrightarrow{=} g(x))$ for every $x : X$. Then for all x, x' in X, $e : x \xrightarrow{=} x'$, and $i : f(x) \xrightarrow{=} g(x)$ we have:*

$$\mathrm{trp}^Z_e(i) \xrightarrow{=} \mathrm{ap}_g(e) \cdot i \cdot \mathrm{ap}_f(e)^{-1}.$$

xca:trp-in-a/x=b/x

Exercise 2.14.4. Implement Construction 2.14.3 in the following special cases, where $Y \equiv X$ and a, b are elements of X:

trp-in-a=x
trp-in-x=a
trp-in-x=x

(1) $\mathrm{trp}_e^{x\mapsto a\xrightarrow{=}b}(i) \xrightarrow{=} i$;

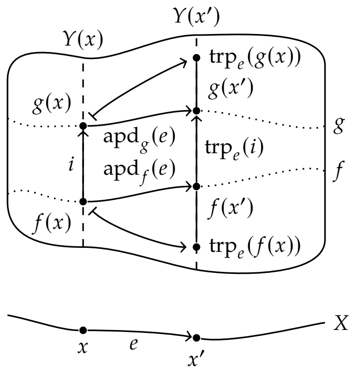
(2) $\mathrm{trp}_e^{x\mapsto a\xrightarrow{=}x}(i) \xrightarrow{=} e \cdot i$;

(3) $\mathrm{trp}_e^{x\mapsto x\xrightarrow{=}b}(i) \xrightarrow{=} i \cdot e^{-1}$;

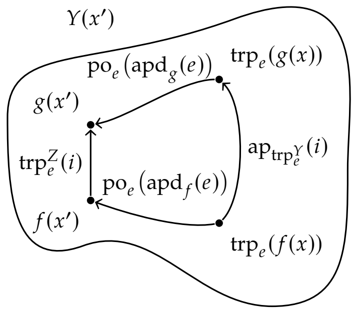
(4) $\mathrm{trp}_e^{x\mapsto x\xrightarrow{=}x}(i) \xrightarrow{=} e \cdot i \cdot e^{-1}$ (also called *conjugation*). ⌟

There is also a dependent version of Construction 2.14.3, which is again proved by induction on e.[42]

[42] We picture this in two stages. First, we show the fiberwise situation as follows:



Here, there's not room to show all that's going on in the fiber $Y(x')$, so we illustrate that as follows:


$Y(x')$
$\mathrm{po}_e(\mathrm{apd}_g(e))$
$\mathrm{trp}_e(g(x))$
$g(x')$
$\mathrm{trp}^Z_e(i)$
$\mathrm{ap}_{\mathrm{trp}^Y_e}(i)$
$\mathrm{po}_e(\mathrm{apd}_f(e))$
$f(x')$
$\mathrm{trp}_e(f(x))$

lem:trp-in-fx=Ygx

Construction 2.14.5. *Let $X, Y(x)$ be types and $f(x), g(x) : Y(x)$ for all $x : X$. Let $Z(x) :\equiv (f(x) \xrightarrow{=} g(x))$, with the identification in $Y(x)$, for every $x : X$. Then for all x, x' in X, $e : x \xrightarrow{=} x'$, and $i : f(x) \xrightarrow{=} g(x)$ we have:*

$$\mathrm{trp}^Z_e(i) \xrightarrow{=} \mathrm{po}_e\big(\mathrm{apd}_g(e)\big) \cdot \mathrm{ap}_{\mathrm{trp}^Y_e}(i) \cdot \mathrm{po}_e\big(\mathrm{apd}_f(e)\big)^{-1}.$$

The following construction will be used later in the book.

def:Dan's-lemma

Definition 2.14.6. Let $X, Y(x)$ be types and $f(x) : Y(x)$ for all $x : X$. Given elements $x, x' : X$ and a path $p : x \xrightarrow{=} x'$, we define an equivalence $(f(x) \underset{e}{\xrightarrow{=}} f(x')) \simeq (f(x) \xrightarrow{=} f(x))$. We do this by inducion on p, using Definition 2.7.1, thereby reducing to the case $(f(x) \xrightarrow{=} f(x)) \simeq (f(x) \xrightarrow{=} f(x))$, which we solve in the canonical way as before. ⌟

sec:props-sets-grpds

2.15 *Propositions, sets and groupoids*

Let P be a type. The property that P has at most one element may be expressed by saying that any two elements are equal. Hence it is encoded by $\prod_{a,b:P}(a \xrightarrow{=} b)$. We shall call a type P with that property a *proposition*, and its elements will be called *proofs* of P. We will use them for doing logic in type theory. The reason for doing so is that the most relevant thing about a logical proposition is whether it has a proof or not. It is therefore reasonable to require for any type representing a logical proposition that all its members are equal.

Suppose P is a proposition. Then English phrases such as "P holds", "we know P", and "we have shown P", will all mean that we have an element of P. We will not use such phrases for types that are not propositions, nor will we discuss knowing P conditionally with a phrase such as "whether P". Similarly, if "Q" is the English phrase for a statement encoded by the proposition P, then the English phrases "Q holds", "we know Q", and "we have shown Q", will all mean that we have an element of P.

Typically, mathematical properties expressed in English as *adjectives* will be encoded by types that are propositions, for in English speech, when you assert that a certain adjective holds, you are simply asserting it, and not providing further information. Examples: the number 6 is *even*; the number 7 is *prime*; the number 28 is *perfect*; consider a *regular* pentagon; consider an *isosceles* triangle.

Sometimes adjectives are used in mathematics, not to refer to properties of an object, but to modify the meaning of a noun, producing a different noun phrase denoting a different mathematical concept. For example, a *directed* graph is a graph, each of whose edges is given a bit of additional information: a direction in which it points. Other examples: *differentiable* manifold; *bipartite* graph; *vector* space; *oriented* manifold.

Let X be a type. If for any $x : X$ and any $y : X$ the identity type $x \xrightarrow{=} y$ is a proposition, then we shall say that X is a *set*. The reason for doing so is that the most relevant thing about a set is which elements it has; distinct identifications of equal elements are not relevant. Alternatively, we shall say that X is a 0-*type*.[43]

[43]Sets are thought to consist of points. Points are entities of dimension 0, which explains why the count starts here. One of the contributions of Vladimir Voevodsky is the extension of the hierarchy downwards, with the notion of proposition, including logic in the same hierarchy. Some authors therefore call propositions (-1)-*types*, and they call contractible types (-2)-*types*.

DEFINITION 2.15.1. Let A be a *set*, as defined above, and let a and b be elements of A. We write $a = b$ as alternative notation for the type $a \xrightarrow{=} b$. Formally, we define it as follows.

$$a = b :\equiv a \xrightarrow{=} b$$

The type $a = b$ is called an *equation*. When it has an element, we say that a and b are *equal*. ⌟

Equations are propositions, so we can speak of them being true or false, and we may use them after the words *if*, *since*, *whether*, and *because* in a sentence. In set theory, everything is a set and all equations $a = b$ are propositions; our definition of $a = b$ is designed to make the transition from set theory to type theory minimally disconcerting.

(Good motivation for the form of the equal sign in the notation $a = b$ is provided by a remark made by Robert Recorde in 1557 in the *Whetstone of Witte*[44]: "And to avoid the tedious repetition of these words *is equal to*, I will set, as I do often in work use, a pair of parallels, or twin lines of one length, thus: =, because no two things can be more equal."[45] In fact, the remark of Recorde presages the approach described in this book, for although those two little lines are congruent, they were not considered to be equal traditionally, since they are in different places, whereas they may be considered to be equal in the presence of univalence, which converts congruences to identities.)

[44]Robert Recorde and John Kingston. *The whetstone of witte: whiche is the seconde parte of Arithmetike, containyng thextraction of rootes, the cossike practise, with the rule of equation, and the woorkes of surde nombers*. Imprynted at London: By Ihon Kyngstone, 1557. URL: https://archive.org/details/TheWhetstoneOfWitte.

[45]And to auoide the tediouſe repetition of theſe woordeſ : iſ equalle to : I will ſette aſ I doe often in woorke vſe, a paire of paralleleſ, or Gemowe lineſ of one lengthe, thuſ: $=\!=\!=$, bicauſe noe .2. thyngeſ, can be moare equalle.

Let X be a type. If for any $x : X$ and any $y : X$ the identity type $x \xrightarrow{=} y$ is a set, then we shall say that X is a *groupoid*, also called a 1-*type*.

The pattern continues. If for any $n : \mathbb{N}$, any $x : X$, and any $y : X$ the identity type $x \xrightarrow{=} y$ is an n-*type*, then we shall say that X is an $(n+1)$-*type*. If X is an n-type, we also say that X is n-*truncated*.

We prove that every proposition is a set, from which it follows by induction that every n-type is an $(n+1)$-*type*.

lem:prop-is-set

LEMMA 2.15.2. *Every type that is a proposition is also a set.*

Proof. Let X be a type and let $f : \prod_{a,b:X}(a \xrightarrow{=} b)$. Let $a, b, c : X$ and let $P(x)$ be the type $a \xrightarrow{=} x$ depending on $x : X$. Then $f(a,b) : P(b)$ and $f(a,c) : P(c)$. By path induction we prove for all $q : b \xrightarrow{=} c$ that $q \cdot f(a,b) \xrightarrow{=} f(a,c)$. For this it suffices to verify that $\mathrm{refl}_b \cdot f(a,b) \xrightarrow{=} f(a,b)$, which follows immediately. So q is equal to $f(a,c)\cdot f(a,b)^{-1}$ which doesn't depend on q, so all such q are equal. Hence X is a set. □

A more interesting example of a set is Bool.

lem:isset-bool

LEMMA 2.15.3. Bool *is a set.*

Proof. The following elegant, self-contained proof is due to Simon Huber. For proving $p \xrightarrow{=} q$ for all $b, b' : \mathrm{Bool}$ and $p, q : b \xrightarrow{=} b'$, it suffices (by induction on q) to show $p \xrightarrow{=} \mathrm{refl}_b$ for all $b : \mathrm{Bool}$ and $p : b \xrightarrow{=} b$. To this end, define by induction on $b, b' : \mathrm{Bool}$, a type $C(b, b', p)$ for all $p : b \xrightarrow{=} b'$, by setting $C(\mathrm{yes}, \mathrm{yes}, p) :\equiv (p \xrightarrow{=} \mathrm{refl}_{\mathrm{yes}})$, $C(\mathrm{no}, \mathrm{no}, p) :\equiv (p \xrightarrow{=} \mathrm{refl}_{\mathrm{no}})$, and arbitrary in the other two cases. By induction on b one proves that $C(b, b, p) \xrightarrow{=} (p \xrightarrow{=} \mathrm{refl}_b)$ for all p. Hence it suffices to prove $C(b, b', p)$ for all $b, b' : \mathrm{Bool}$ and $p : b \xrightarrow{=} b'$. By induction on p this reduces to $C(b, b, \mathrm{refl}_b)$, which is immediate by induction on $b : \mathrm{Bool}$. □

We now collect a number of useful results on propositions.

lem:prop-utils

LEMMA 2.15.4. *Let A be a type, and let P and Q propositions. Let $R(a)$ be a proposition depending on $a : A$. Then we have:*

(1) False *and* True *are propositions;* prop-utils-false-true

(2) *$A \to P$ is a proposition;* prop-utils-codom

(3) *$\prod_{a:A} R(a)$ is a proposition;* prop-utils-pi

(4) *$P \times Q$ is a proposition;* prop-utils-times

(5) *if A is a proposition, then $\sum_{a:A} R(a)$ is a proposition;* prop-utils-sum

(6) *$P \simeq Q$ is a proposition;* prop-utils-eq

(7) *$P \amalg \neg P$ is a proposition.* prop-utils-lem

Proof. (1): If $p, q : \mathrm{False}$, then $p \xrightarrow{=} q$ holds vacuously. If $p, q : \mathrm{True}$, then $p \xrightarrow{=} q$ is proved by double induction, which reduces the proof to observing that $\mathrm{refl}_{\mathrm{triv}} : \mathrm{triv} \xrightarrow{=} \mathrm{triv}$.

(2): If $p, q : A \to P$, then $p \xrightarrow{=} q$ is proved by first observing that p and q are functions which, by function extensionality, are equal if they have equal values $p(x) \xrightarrow{=} q(x)$ in P for all x in A. This is actually the case since P is a proposition.

(3): If $p, q : \prod_{a:A} R(a)$ one can use the same argument as for $A \to P$ but now with *dependent* functions p, q.

(4): If $(p_1, q_1), (p_2, q_2) : P \times Q$, then $(p_1, q_1) \xrightarrow{=} (p_2, q_2)$ is proved componentwise. Alternatively, we may regard this case as a special case of (5).

(5): Given $(a_1, r_1), (a_2, r_2) : \sum_a R(a)$, we must establish that $(a_1, r_1) \xrightarrow{=} (a_2, r_2)$. Combining the map in Definition 2.10.1 with the identity in Definition 2.7.2 yields a map $\left(\sum_{u : a_1 \xrightarrow{=} a_2} \mathrm{trp}^Y_u(r_1) \xrightarrow{=} r_2\right) \to ((a_1, r_1) \xrightarrow{=} (a_2, r_2))$, so it suffices to construct an element in the source of the map. Since A is a proposition, we may find $u : a_1 \xrightarrow{=} a_2$. Since $R(a_2)$ is a proposition, we may find $v : \mathrm{trp}^Y_u(r_1) \xrightarrow{=} r_2$. The pair (u, v) is what we wanted to find.

(6): Using Lemma 2.9.9, $P \simeq Q$ is equivalent to $(P \to Q) \times (Q \to P)$, which is a proposition by combining (2) and (4).

(7): If $p, q : P \amalg \neg P$, then we can distinguish four cases based on inl/inr, see Section 2.8. In two cases we have both P and $\neg P$ and we are done.

In the other two, either $p \equiv \mathrm{inl}_{p'}$ and $q \equiv \mathrm{inl}_{q'}$ with $p', q' : P$, or $p \equiv \mathrm{inr}_{p'}$ and $q \equiv \mathrm{inr}_{q'}$ with $p', q' : \neg P$. In both these cases we are done since P and $\neg P$ are propositions. □

Several remarks can be made here. First, the lemma supports the use of False and True as truth values, and the use of $\to, \prod, \times$ for implication, universal quantification, and conjunction, respectively. Since False is a proposition, it follows by (2) above that $\neg A$ is a proposition for any type A. As noted before, (2) is a special case of (3).

Notably absent in the lemma above are disjunction and existential quantification. This has a simple reason: True $\amalg$ True has two distinct elements $\mathrm{inl}_{\mathrm{triv}}$ and $\mathrm{inr}_{\mathrm{triv}}$, an is therefore *not* a proposition. Similarly, $\sum_{n:\mathbb{N}}$ True has infinitely many distinct elements (n, triv) and is not a proposition. We will explain in Section 2.16 how to work with disjunction and existential quantification for propositions.

The lemma above has a generalization from propositions to n-types which we state without proving. (The proof goes by induction on n, with the lemma above serving as the base case where n is -1.)

lem:level-n-utils

LEMMA 2.15.5. *Let A be a type, and let X and Y be n-types. Let $Z(a)$ be an n-type depending on $a : A$. Then we have*:

level-n-utils-codom

(1) *$A \to X$ is an n-type;*

level-n-utils-pi

(2) *$\prod_{a:A} Z(a)$ is an n-type;*

level-n-utils-times

(3) *$X \times Y$ is an n-type.*

level-n-utils-sum

(4) *if A is an n-type, then $\sum_{a:A} Z(a)$ is an n-type;*

We formalize the definitions from the start of this section.

def:isSet

DEFINITION 2.15.6.

$$\begin{aligned}\mathrm{isProp}(P) &:\equiv \prod_{p,q:P}(p \xrightarrow{=} q)\\ \mathrm{isSet}(S) &:\equiv \prod_{x,y:S} \mathrm{isProp}(x \xrightarrow{=} y) \equiv \prod_{x,y:S}\prod_{p,q:(x \xrightarrow{=} y)}(p \xrightarrow{=} q)\\ \mathrm{isGrpd}(G) &:\equiv \prod_{g,h:G} \mathrm{isSet}(g \xrightarrow{=} h) \equiv \dots\end{aligned}$$ ⌟

lem:isX-is-prop

LEMMA 2.15.7. *For any type A, the following types are propositions*:

(1) isContr(A);

(2) isProp(A);

(3) isSet(A);

(4) isGrpd(A);

(5) *the type that encodes whether A is an n-type, for $n \geq 0$.*

Consistent with that, we will use identifiers starting with "is" only for names of types that are propositions. Examples are isSet(A) and isGrpd(A), and also isEquiv(f).

Proof. Recall that isContr(A) is $\sum_{a:A}\prod_{y:A}(a \xrightarrow{=} y)$. Let (a, f) and (b, g) be elements of the type isContr(A). By Definition 2.10.1, to give an element of $(a, f) \xrightarrow{=} (b, g)$ it suffices to give an $e : a \xrightarrow{=} b$ and an $e' : f \underset{e}{\xrightarrow{=}} g$. For e we can take $f(b)$; for e' it suffices by Definition 2.7.2 to give an $e'' : \mathrm{trp}_e f \overset{=}{\to} g$. Clearly, A is a proposition and hence a set by Lemma 2.15.2. Hence the type of g is a proposition by Lemma 2.15.4(3), which gives us e''.

We leave the other cases as exercises. □

xca:isX-is-prop

Exercise 2.15.8. Make sure you understand that isProp(P) is a proposition, using the same lemmas as for isContr(A). Show that isSet(S) and isGrpd(G) are propositions. ⌟

The following exercise shows that the inductive definition of n-types can indeed start with n as -2, where we have the contractible types.

xca:prop-contractible=

Exercise 2.15.9. Given a type P, show that P is a proposition if and only if $p \overset{=}{\to} q$ is contractible, for any $p, q : P$. ⌟

We now present the notion of a *diagram*. A diagram is a graph whose vertices are types and whose edges are functions. Here is an example.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "p"] & Y \arrow[d, "q"] \\
S \arrow[r, "g"] & T
\end{tikzcd}$$

The information conveyed by this diagram to the reader is that X, Y, S, and T are types, and that f, g, p, and q are functions; moreover, f is of type $X \to Y$, g is of type $S \to T$, p is of type $X \to S$, and q is of type $Y \to T$.

Observe that we can travel through the diagram from X to T by following first the arrow labeled f and then the arrow labelled q. Consequently, the composite function $q \circ f$ is of type $X \to T$.

There is another route from X to T : we could follow first the arrow labeled p and then the arrow labelled g. Consequently, the composite function $g \circ p$ is also of type $X \to T$.

We say that a diagram *is commutative by definition* if, whenever there are two routes from one vertex to another, the corresponding composite functions are equal by definition. For example, in the diagram above, the condition would be that $g \circ p \equiv q \circ f$.

When the function type from any vertex of a diagram to any other vertex of the diagram is a set, then equality of functions is a proposition, and we may consider whether two functions are equal. In that case, we say that a diagram *is commutative* if, whenever there are two routes from one vertex to another, the corresponding composite functions are equal. For example, in the diagram above, the condition would be that $g \circ p = q \circ f$.

There are other sorts of diagrams. For example, identifications may be composed, and thus we may have a diagram of identifications between elements of the same type. For example, suppose W is a type, suppose that x, y, s, and t are elements of W, and consider the following diagram.

$$\begin{tikzcd}
x \arrow[r, "f", "="'] \arrow[d, "p", "\shortparallel"'] & y \arrow[d, "q", "\shortparallel"'] \\
s \arrow[r, "g", "="'] & t
\end{tikzcd}$$

It indicates that f is of type $x \overset{=}{\to} y$, g is of type $s \overset{=}{\to} t$, p is of type $x \overset{=}{\to} s$, and q is of type $y \overset{=}{\to} t$. We may also consider whether such a diagram is commutative by definition, or, in the case where all the identity types are sets, is commutative.

2.16 *Propositional truncation and logic*

sec:prop-trunc

As explained in Section 2.15, the type formers $\to, \prod, \times$ can be used with types that are propositions for the logical operations of implication, universal quantification, and conjunction, respectively. Moreover, True and False can be used as truth values, and $\neg$ can be used for negation. We have also seen that $\amalg$ and Σ can lead to types that are not propositions, even though the constituents are propositions. This means we are still lacking disjunction ($P \vee Q$) and existence ($\exists_{x:X} P(x)$) from the standard repertoire of logic, as well as the notion of *non-emptiness* of a type. In this section we explain how to implement these three notions.

To motivate the construction that follows, consider non-emptiness of a type T. In order to be in a position to encode the mathematical assertion expressed by the English phrase "T is non-empty", we will need a proposition P. The proposition P will have to be constructed somehow from T. Any element of T should somehow give rise to an element of P, but, since all elements of propositions are equal to each other, all elements of P arising from elements of T should somehow be made to equal each other. Finally, any proposition Q that is a consequence of having an element of T should also be a consequence of P.

We define now an operation called propositional truncation,[46] that enforces that all elements of a type become equal.

[46]The name "truncation" is slightly misleading since it suggests leaving something out, whereas the correct intuition is one of adding identifications so everything becomes equal.

def:prop-trunc

Definition 2.16.1. Let T be a type. The *propositional truncation* of T is the type $\|T\|$ defined by the following constructors:

(1) an *element* constructor $|t| : \|T\|$ for all $t : T$;

(2) an *identification* constructor providing an identity of type $x \xrightarrow{=} y$ for all $x, y : \|T\|$.

The identification constructor ensures that $\|T\|$ is a proposition. The induction principle states that, for any family of propositions $P(x)$ parametrized by a variable $x : \|T\|$, in order to prove $\prod_{x:\|T\|} P(x)$, it suffices to prove $\prod_{t:T} P(|t|)$. In other words, in order to define a function $f : \prod_{x:\|T\|} P(x)$, it suffices to give a function $g : \prod_{t:T} P(|t|)$. Moreover, the function f will satisfy $f(|t|) \equiv g(t)$ for all $t : T$. ⌟

Consider the special case where the family $P(x)$ is constant. We see that any function $g : T \to P$ to a proposition P yields a (unique) function $f : \|T\| \to P$ satisfying $f(|t|) \equiv g(t)$ for all $t : T$.[47] A useful consequence of this recursion principle is that, for any proposition P, precomposition with $|\_|$ is an equivalence of type

$$(\|T\| \to P) \quad \xrightarrow{\simeq} \quad (T \to P).$$

[47]Given $t, t' : T$, we have an identification of type $|t| \xrightarrow{=} |t'|$. The existence of the function g implies that we have an identification of type $g(|t|) \xrightarrow{=} g(|t'|)$, and hence an identification of type $f(t) \xrightarrow{=} f(t')$. Thus a necessary condition for the existence of g is the existence of identifications of type $f(t) \xrightarrow{=} f(t')$. That justifies the the hypothesis that P is proposition.

This is called *the universal property of propositional truncation*.

def:non-empty

Definition 2.16.2. Let T be a type. We call T *non-empty* if we have an element of $\|T\|$.[48] ⌟

[48]We may alternatively say that T is *inhabited*, in order to avoid confusion with the concept of T *not being empty*, which would be represented by the proposition $\neg(T \xrightarrow{=} \emptyset)$, which is equivalent to $\neg\neg T$.

Now that propositional truncation is available, we are ready to define logical disjunction and existence.

Definition 2.16.3. Given propositions P and Q, define their *disjunction* by $(P \vee Q) :\equiv \|P \amalg Q\|$. It expresses the property that P is true or Q is true. ⌟

Definition 2.16.4. Given a type X and a family $P(x)$ of propositions parametrized by a variable x of type X, define a proposition that encodes the property that there exists a member of the family for which the property is true by $(\exists_{x:X} P(x)) :\equiv \|\sum_{x:X} P(x)\|$. It expresses the property that there *exists* an element $x : X$ for which the property $P(x)$ is true; the element x is not given explicitly. ⌟

The following logical quantifier could have been defined earlier, since it doesn't use propositional truncation. We present it now, for completeness.

Definition 2.16.5. Given a type X and a family $P(x)$ of propositions parametrized by a variable x of type X, define a proposition that encodes the property that there exists a *unique* member of the family for which the property is true by the proposition $(\exists!_{x:X} P(x)) :\equiv \mathrm{isContr}(\sum_{x:X} P(x))$. ⌟

Exercise 2.16.6. Given $x : \|T\|$, prove that $\exists_{t:T}(x = |t|)$. ⌟

xca:prop-set-trivia-1

Exercise 2.16.7. Suppose P is a proposition. Produce an equivalence of type $P \xrightarrow{\simeq} \|P\|$. ⌟

The exercise above us to easily convert elements of type $\|P\|$ to elements of type P when P is a proposition.

def:connected

Definition 2.16.8. Let A be a type. If $a : A$, then the subtype $A_{(a)} :\equiv \sum_{x:A} \|x \xrightarrow{=} a\|$ is called the *connected component* of a in A. We say that elements x, y of A are *in the same component* of A if $\|x \xrightarrow{=} y\|$, for then $A_{(x)} = A_{(y)}$. The type A is called *connected*[49] if it is non-empty with all elements in the same component. Formally, this property is encoded by the following proposition.

$$\mathrm{isConn}(A) :\equiv \|A\| \times \prod_{x,y:A} \|x \xrightarrow{=} y\|. \quad ⌟$$

[49] In Definition 2.22.4 below we will define the *set of connected components* of a type.

Note that the empty type $\emptyset$ is *not* connected.

One can view being connected as a weak form of being contractible – without direct access to a center and to identifications of elements.

xca:component-connected

Exercise 2.16.9. Show that the component of a in A is connected. Show that equal elements have the same *propositional* properties, that is, for any predicate $P : A \to \mathcal{U}$, $P(x)$ is equivalent to $P(y)$ for any $x, y : A$ with $x = y$. ⌟

xca:prop-set-trivia-2

Exercise 2.16.10. Show that any connected set is contractible. ⌟

xca:connected-trivia

Exercise 2.16.11. Let A be a connected type, and suppose that $a \xrightarrow{=} a$ is a proposition for every $a : A$. Show that A is contractible. ⌟

In the following definition we introduce the adverb *merely*, which serves as a quicker way to say *the propositional truncation of* in English speech.

def:merely

Definition 2.16.12. What we mean by *merely* constructing an element of a type T is constructing an element of $\|T\|$. ⌟

For example, a type is non-empty if it *merely has an element*, and a type is connected if any two elements can be *merely identified* with each other.

We now make precise the meaning of the word *equivalent*, which was introduced earlier.

DEFINITION 2.16.13. If X and Y are types, then the phrase "X and Y are equivalent" means that an equivalence between them can be *merely* constructed. It is encoded by the type $\|X \xrightarrow{\simeq} Y\|$. ⌟

2.17 *More on equivalences; surjections and injections*

sec:more-on-equivalences

In this section we collect a number of useful results on equivalences.

Consider the function $f : \mathbb{1} \to \mathbb{2}$ sending 0 to 0. The fibers of f at 0 and 1 are equivalent to True and False. Hence f is not an equivalence, since False is not contractible. Observe that both fibers are propositions, that is, contain at most one element.

As a function between sets f is an injection (one-to-one), but not a surjection. We need these important concepts for types in general. We define them as close as possible to their usual meaning in set theory: a function from A to B is surjective if the preimage of any $b : B$ is non-empty, and injective if such preimages contain at most one element. This motivates the following definitions.

def:surjection

DEFINITION 2.17.1. A function $f : A \to B$ is a *surjection*, or is *surjective*, if for all $b : B$ there exists an $a : A$ such that $b \overset{=}{\to} f(a)$, that is, $\exists_{a:A} b \overset{=}{\to} f(a)$.[50] ⌟

[50] A function $f : A \to B$ is a *split surjection* if for all $b : B$ there (purely) is an $a : A$ with $b \overset{=}{\to} f(a)$, in other words, we have a function of type $\prod_{b:B} \sum_{a:A} b \overset{=}{\to} f(a)$. This is equivalent to saying we have a function $g : B \to A$ such that $f \circ g \overset{=}{\to} \mathrm{id}_B$ (such a g is called a *section* of f).

def:injection

DEFINITION 2.17.2. A function $f : A \to B$ is an *injection*, or is *injective*, if $f^{-1}(b)$ is a proposition for all $b : B$. The property of being an injection is encoded by the type $\mathrm{isInj}(f) :\equiv \prod_{b:B} \mathrm{isProp}(f^{-1}(b))$. ⌟

xca:inj-sets

EXERCISE 2.17.3. Show that if A, B are sets, then a function $f : A \to B$ is injective if and only if $f(a) \overset{=}{\to} f(a')$ implies $a \overset{=}{\to} a'$ for all a, a'. ⌟

lem:inj+surj

LEMMA 2.17.4. *For all types A, B, a function $f : A \to B$ is an equivalence if and only if f is an injection and a surjection.*

Proof. If $f : A \to B$ is an equivalence, then all fibers are contractible, so f is both an injection and a surjection. Conversely, if f is both injective and surjective, we show that $f^{-1}(b)$ is contractible, for each $b : B$. Being contractible is a proposition, so by Definition 2.16.1 we can drop the truncation in $\|\sum_{a:A} b \overset{=}{\to} f(a)\|$. Now apply injectivity.[51] □

[51] This argument applies generally: Any non-empty proposition is contractible.

If the types A and B in the above lemma are *sets*, then we call equivalences between A and B also *bijections*.

cor:inj+connected

COROLLARY 2.17.5. *Let A, B be types such that A is non-empty and B is connected. Then any injection $f : A \to B$ is an equivalence.*

Proof. By Lemma 2.17.4 it suffices to show that f is surjective. This is a proposition, so by Definition 2.16.1 and $\|A\|$ we may assume $a : A$, so $f(a) : B$. By $\prod_{x,y:B} \|x \overset{=}{\to} y\|$ we now get that all preimages under f are non-empty. □

lem:whenisbasespaceconnected

LEMMA 2.17.6. *Let $f : X \to Y$ be a surjective map from a connected type X. Then Y is connected too.*

Proof. For any map $f : X \to Y$ between arbitrary types, if $y, y' : Y$ and we are given $x, x' : X$, $p : y \overset{=}{\to} f(x)$, $p' : y' \overset{=}{\to} f(x')$ and $q : x \overset{=}{\to} x'$, then we have an identity between y and y' given by the composite

$$y \xrightarrow[p]{=} f(x) \xrightarrow[f(q)]{=} f(x') \xrightarrow[p'^{-1}]{=} y'.$$

Now the lemma follows by eliminating the propositional truncations in the assumptions, using that the conclusion is a proposition. □

con:fib-vs-path

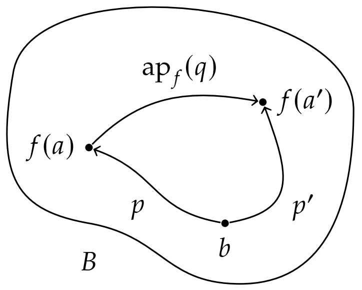
CONSTRUCTION 2.17.7. *For every $f : A \to B$, $b : B$, and $z, z' : f^{-1}(b)$, there is an equivalence*

{eq:fib_vs_path}

$$(z \overset{=}{\to} z') \simeq \mathrm{ap}_f^{-1}(\mathrm{snd}\, z' \cdot \mathrm{snd}\, z^{-1}). \tag{2.17.1}$$

Implementation of Construction 2.17.7. We can construct this equivalence for $z \equiv (a, p)$ and $z' \equiv (a', p')$, where $a, a' : A$, $p : b \overset{=}{\to} f(a)$ and $p' : b \overset{=}{\to} f(a')$, as the composition

$$\begin{aligned}(z \overset{=}{\to} z') &\equiv \big((a, p) \overset{=}{\to} (a', p')\big)\\ &\simeq \sum_{q : a \overset{=}{\to} a'} p \overset{=}{\underset{q}{\to}} p'\\ &\simeq \sum_{q : a \overset{=}{\to} a'} \mathrm{ap}_f(q) \cdot p \overset{=}{\to} p'\\ &\simeq \sum_{q : a \overset{=}{\to} a'} p' \cdot p^{-1} \overset{=}{\to} \mathrm{ap}_f(q)\\ &\equiv \mathrm{ap}_f^{-1}(p' \cdot p^{-1}).\end{aligned}$$



The second equivalence relies on Definition 2.7.2 and Construction 2.14.3. □

lem:inj-ap

LEMMA 2.17.8. *A function $f : A \to B$ is an injection if and only if each induced function $\mathrm{ap}_f : (a \overset{=}{\to} a') \to (f(a) \overset{=}{\to} f(a'))$ is an equivalence, for all $a, a' : A$.*[52]

[52] *Warning*: If A and B are sets, then each ap_f is an equivalence if and only if we have the implication $(f(a) \overset{=}{\to} f(a')) \to (a \overset{=}{\to} a')$, but this is in general not sufficient.

Proof. It follows directly from (2.17.1) that if ap_f is an equivalence, then $f^{-1}(b)$ is a proposition, as all its identity types are contractible.

On the other hand, if we fix $a, a' : A$ and $p : f(a) \overset{=}{\to} f(a')$, then (2.17.1) applied to $b :\equiv f(a)$, $z :\equiv (a, \mathrm{refl}_{f(a)})$ and $z' :\equiv (a', p)$, gives $\mathrm{ap}_f^{-1}(p) \simeq (z \overset{=}{\to} kz')$, which shows that if each $f^{-1}(b)$ is a proposition, then ap_f is an equivalence. □

cor:fib-vs-path

COROLLARY 2.17.9. *Let A and B be types and let $f : A \to B$ be a function. Then we have:*

set-fib-vs-path

(1) *All fibers of f are $n + 1$-types if and only if all fibers of each map induced by f on identity types are n-types;*

conn-fib-vs-path

(2) *If A and B are connected, then f is an equivalence if and only if each map induced by f on identity types is an equivalence;*

conn-fib-vs-path-point

(3) *If A and B are connected and $a : A$, then f is an equivalence if and only if $\mathrm{ap}_f : (a \overset{=}{\to} a) \to (f(a) \overset{=}{\to} f(a))$ is an equivalence.*

Proof. (1) When n is -2 this is Lemma 2.17.8 and the proof for $n \geq -1$ is similar. (2) By Lemma 2.17.8 and Corollary 2.17.5. (3) By (2) and Exercise 2.16.9. □

xca:sum-equivalences

Exercise 2.17.10. Let $A, B : \mathcal{U}$, $F : A \to \mathcal{U}$ and $G : B \to \mathcal{U}$, and $f : A \simeq B$ and $g : \prod_{a:A}(F(a) \simeq G(f(a)))$. Give an equivalence from $\sum_{a:A} F(a)$ to $\sum_{b:B} G(b)$. (An important special case is $F \equiv G \circ f$.) $\lrcorner$

Another application of propositional truncation is the notion of image.

def:prop-image

Definition 2.17.11. Let A, B be types and let $f : A \to B$. We define the *image* of f as

$$\mathrm{im}(f) :\equiv \sum_{y:B} \exists_{x:A}(y \overset{=}{\to} f\,x). \qquad \lrcorner$$

Note that $(\exists_{x:A}(y \overset{=}{\to} f\,x)) \equiv \|f^{-1}(y)\|$, the propositional truncation of the fiber. For this reason, $\mathrm{im}(f)$ is called the *propositional* image. Later we will meet other notions of image, based on other truncation operations.

xca:unique-fact-image

Exercise 2.17.12. Show that the image of $f : A \to B$ induces a factorization $f \overset{=}{\to} i \circ p$

$$\begin{tikzcd} A \arrow[rr, "f"] \arrow[dr, "p"'] & & B \\ & \mathrm{im}(f) \arrow[ur, "i"'] & \end{tikzcd}$$

where p is surjective and i is injective, and that each such factorization is equivalent to the image factorization. $\lrcorner$

xca:all-prop-image

Exercise 2.17.13. Let $f : A \to B$ for A and B types, and let $P(b)$ be a proposition depending on $b : B$. Show that $\prod_{z\,:\,\mathrm{im}(f)} P(\mathrm{fst}(z))$ if and only if $\prod_{a:A} P(f(a))$. $\lrcorner$

sec:decidability

2.18 *Decidability, excluded middle and propositional resizing*

Recall from Lemma 2.15.4(7) that $P \amalg \neg P$ is a proposition whenever P is a proposition.

def:decidability

Definition 2.18.1. A proposition P is called *decidable* if $P \amalg \neg P$ holds. $\lrcorner$

In traditional mathematics, it is usually assumed that every proposition is decidable. This is expressed by the following principle, commonly abbreviated LEM.

pri:lem

Principle 2.18.2 (Law of Excluded Middle). For every proposition P, the proposition $P \amalg \neg P$ holds. $\lrcorner$

(The "middle" ground excluded by this principle is the possibility that there is a proposition that it neither true nor false.)

Type theory is born in a constructivist tradition which aims at developing as much mathematics as possible without assuming the Law of Excluded Middle.[53] Following this idea, we will explicitly state whenever we are assuming the Law of Excluded Middle.

[53]Besides any philosophical reasons, there are several pragmatic reasons for developing constructive mathematics. One is that proofs in constructive mathematics can be executed as programs, and another is that the results also hold in non-standard models, for instance a model where every type has a topological structure, and all constructions are continuous. See also Footnote 13.

xca:lem-prop

Exercise 2.18.3. Show that the Law of Excluded Middle is equivalent to asserting that the map $(\mathrm{yes} \overset{=}{\to} \_) : \mathrm{Bool} \to \mathrm{Prop}$ is an equivalence. $\lrcorner$

A useful consequence of the Law of Excluded Middle is the so called principle of "proof by contradiction": to prove a proposition P, assume its negation $\neg P$ and derive a contradiction. Without the Law of Excluded Middle, this proves only the double negation of P, that is $\neg\neg P$. However, with the Law of Excluded Middle, one can derive P from the latter: indeed, according to the Law of Excluded Middle, either P or $\neg P$ holds; but $\neg P$ leads to a contradiction by hypothesis, making P hold necessarily.

xca:dne-lem

EXERCISE 2.18.4. Show that, conversely, LEM follows from the principle of *double-negation elimination*: For every proposition P, if $\neg\neg P$, then P holds. ⌟

REMARK 2.18.5. We will later encounter a weaker version of the Law of Excluded Middle, called the Limited Principle of Omniscience (Principle 3.6.16), which is often enough.[54] ⌟

[54] As the naming indicates, we can think of the Law of Excluded Middle itself as an omniscience principle, telling us for every proposition P, whether P is true or false. It was this interpretation of the Law of Excluded Middle that led Brouwer to reject it in his 1908 paper on *De onbetrouwbaarheid der logische principes*.[55]

[55] Mark van Atten and Göran Sundholm. "L.E.J. Brouwer's 'Unreliability of the Logical Principles A New Translation, with an Introduction". In: *History and Philosophy of Logic* 38.1 (2017), pp. 24–47. DOI: 10.1080/01445340.2016.1210986. arXiv: 1511.01113.

Sometimes we make use of the following, which is another consequence of the Law of Excluded Middle:

pri:prop-resizing

PRINCIPLE 2.18.6 (Propositional Resizing). For any pair of nested universes $\mathcal{U} : \mathcal{U}'$, the inclusion $\mathrm{Prop}_{\mathcal{U}} \to \mathrm{Prop}_{\mathcal{U}'}$ is an equivalence. ⌟

xca:lem-prop-sizing

EXERCISE 2.18.7. Show that if the Law of Excluded Middle holds for all propositions, then propositional resizing holds. ⌟

sec:replacement

2.19 *The replacement principle*

In this section we fix a universe $\mathcal{U}$. We think of types $A : \mathcal{U}$ as *small* compared to arbitrary types, which are then *large* in comparison.[56] Often we run into types that are not in $\mathcal{U}$ (small) directly, but are nevertheless equivalent to types in $\mathcal{U}$.

[56] The terminology *small/large* is also known from set theory, where classes are large collections, and sets are small collections.

def:ess-loc-small

DEFINITION 2.19.1. We say that a type A is *essentially $\mathcal{U}$-small* if there (purely) is a type $X : \mathcal{U}$ and an equivalence $A \simeq X$. And A is *locally $\mathcal{U}$-small* if all its identity types are essentially $\mathcal{U}$-small. ⌟

Note that $\sum_{X:\mathcal{U}}(A \simeq X)$ is a proposition by the univalence axiom for $\mathcal{U}$. Of course, any $A : \mathcal{U}$ is essentially $\mathcal{U}$-small, and any essentially $\mathcal{U}$-small type is locally $\mathcal{U}$-small.

To show that a type is locally $\mathcal{U}$-small we have to give a reflexive relation $\mathrm{Eq}_A : A \to A \to \mathcal{U}$ that induces, by path induction, a family of equivalences $(x \xrightarrow{=} y) \simeq \mathrm{Eq}_A\, x\, y$.

EXERCISE 2.19.2. Show that $\mathcal{U}$ is locally $\mathcal{U}$-small, and investigate the closure properties of essentially and locally $\mathcal{U}$-small types. (For instance, show that if $A : \mathcal{U}$ and $B(x)$ is a family of locally $\mathcal{U}$-small types parametrized by $x : A$, then $\prod_{x:A} B(x)$ is locally $\mathcal{U}$-small.) ⌟

REMARK 2.19.3. Note that propositional resizing (Principle 2.18.6) equivalently says that any proposition is essentially $\mathcal{U}$-small, where we may take $\mathcal{U}$ to be the smallest universe $\mathcal{U}_0$. When we assume this, we get that any set is locally $\mathcal{U}_0$-small. ⌟

We will make use of the following (recall the definition of the image, Definition 2.17.11):

pri:replacement

PRINCIPLE 2.19.4 (Replacement). For any map $f : A \to B$ from an essentially $\mathcal{U}$-small type A to a locally $\mathcal{U}$-small type B, the image $\mathrm{im}(f)$ is essentially $\mathcal{U}$-small. ⌟

This is reminiscent of the replacement principle of set theory which states that for a large (class-sized) function with domain a small set and codomain the class V of all small sets, the image is again a small set. This follows from our replacement principle, assuming propositional resizing, or the even stronger principle of the excluded middle.

The replacement principle can be proved using the join construction of the image, cf. Rijke[57], which uses as an assumption that the universes

[57] Egbert Rijke. *The join construction*. 2017. arXiv: 1701.07538.

are closed under pushouts.[58]

[58] Pushouts are certain higher inductive types that suffice to construct all the higher inductive types that we need, but we don't actually need them in this book.

xca:comp-loc-small-ess-small

Exercise 2.19.5. Show that the replacement principle implies that for any locally $\mathcal{U}$-small type A, and any element $a : A$, the connected component $A_{(a)}$ is essentially $\mathcal{U}$-small. ⌟

Another consequence is that the type of finite sets, which we'll define below in Definition 2.24.5, is essentially small.

2.20 *Predicates and subtypes*

sec:subtype

In this section, we consider the relationship between predicates on a type T and subtypes of T. The basic idea is that the predicate tells whether an element of T should belong to the subtype, and the predicate can be recovered from the subtype by asking whether an element of T is in it.

def:predicate

Definition 2.20.1. Let T be a type and let $P(t)$ be a family of types parametrized by an variable $t : T$, such that $P(t)$ is a proposition. Then we call P a *predicate* on T.[59] If $P(t)$ is a decidable proposition, then we say that P is a *decidable predicate* on T. ⌟

[59] Note that giving a predicate on T is equivalent to giving a map $Q : T \to \mathrm{Prop}_{\mathcal{U}}$ for a suitable universe $\mathcal{U}$, and we sometimes say that Q itself is the predicate.

By Exercise 2.18.3, the decidable predicates P on T correspond uniquely to the characteristic functions $\chi_P : T \to \mathrm{Bool}$.

We recall from Definition 2.17.2 the notion of *injection*, which will be key to saying what a *subtype* is.

def:subtype

Definition 2.20.2. A *subtype* of a type T is a type S together with an injection $f : S \to T$. Selecting a universe $\mathcal{U}$ as a repository for such types S allows us to introduce the type of subtypes of T in $\mathcal{U}$ as follows.

$$\mathrm{Sub}_T^{\mathcal{U}} :\equiv \sum_{S:\mathcal{U}} \sum_{f:S\to T} \mathrm{isInj}(f).$$

We may choose to leave the choice of $\mathcal{U}$ ambiguous, in which case we will write Sub_T for $\mathrm{Sub}_T^{\mathcal{U}}$. ⌟

lem:subtype-eq-=

Lemma 2.20.3. *Let T be a type and P a predicate on T. Consider $\sum_{t:T} P(t)$ and the corresponding projection map* $\mathrm{fst} : T_P :\equiv \left(\sum_{t:T} P(t)\right) \to T$. *Then* $\mathrm{ap}_{\mathrm{fst}} : ((x_1, p_1) \xrightarrow{=} (x_2, p_2)) \to (x_1 \xrightarrow{=} x_2)$ *is an equivalence, for any elements (x_1, p_1) and (x_2, p_2) of T_P.*

Proof. We apply Lemma 2.9.9. Consider $q : x_1 \xrightarrow{=} x_2$. By induction on q we get that each $p_1 \xrightarrow[q]{=} p_2$ is contractible, say with center c_q. We show that mapping q to $\overline{(q, c_q)}$ defines an inverse of $\mathrm{ap}_{\mathrm{fst}}$, applying Lemma 2.10.3 and the remarks after its proof. These give $\mathrm{ap}_{\mathrm{fst}} \overline{(q, c_q)} \xrightarrow{=} q$ for all $q : x_1 \xrightarrow{=} x_2$. Also, for any $r : (x_1, p_1) \xrightarrow{=} (x_2, p_2)$, $r \xrightarrow{=} \overline{(\mathrm{fst}(r), \mathrm{snd}(r))}$. The latter pair is equal to $\overline{(\mathrm{ap}_{\mathrm{fst}(r)}, c)}$ for any c in the contractible type $p_1 \xrightarrow[\mathrm{fst}(r)]{=} p_2$. □

Combined with Lemma 2.17.8, this gives that fst is an injection. Hence, given a predicate P on T, the *subtype* of T characterized by P is defined as $T_P :\equiv \sum_{t:T} P(t)$, together with the injection $\mathrm{fst} : T_P \to T$.

The above lemma has other important consequences.

cor:subtype-same-level

Corollary 2.20.4. *For each natural number n, if T is a n-type, then T_P is also a n-type.*

In particular, if T is a set, then T_P is again a set; we may denote this subset by $\{\, t : T \mid P(t) \,\}$.

rem:subtype-convention

REMARK 2.20.5. Another important consequence of Lemma 2.20.3 is that we can afford not to distinguish carefully between elements (t, p) of the subtype T_P and elements t of type T for which the proposition $P(t)$ holds. We will hence often silently coerce from T_P to T via the first projection, and if $t : T$ is such that $P(t)$ holds, we'll write $t : T_P$ to mean any pair (t, p) where $p : P(t)$, since when $P(t)$ holds, the type $P(t)$ is contractible. ⌟

Given a set A and a function $\chi_B : A \to \text{Bool}$, Lemma 2.15.3 yields that $\chi_B(a) \xrightarrow{=} \text{yes}$ is a proposition, and we can form the subset $\{\, a : A \mid \chi_B(a) \xrightarrow{=} \text{yes} \,\}$. However, not every subset as in Definition 2.20.2 can be given through a $\chi_B : A \to \text{Bool}$. As proved in Section 2.12.1, any element of Bool is equal to yes or to no.

If $P : A \to \mathcal{U}$ is a decidable predicate, then we can define $\chi_P : A \to \text{Bool}$ by induction (actually, only case distinction) on $p : P(a)$, setting $\chi_P(a) \xrightarrow{=} \text{yes}$ if $p \equiv \text{inl}\_$ and $\chi_P(a) \xrightarrow{=} \text{no}$ if $p \equiv \text{inr}\_$. We will often use a characteristic function $T \to \text{Bool}$ to specify a decidable predicate on a type T.

xca:decidability

EXERCISE 2.20.6. Show that $f(t) \xrightarrow{=} \text{yes}$ is a decidable predicate on T, for any type T and function $f : T \to \text{Bool}$. Show $(P \simeq \text{True}) \amalg (P \simeq \text{False})$ for every decidable proposition P. ⌟

We've seen how to make a subtype from a predicate. Conversely, from a subtype of T given by the injection $f : S \to T$, we can form a predicate $P_f : T \to \text{Prop}$ defined by $P_f(x) :\equiv f^{-1}(x)$. We shall see in Lemma 2.25.4, that these operations form an equivalence between predicates on T and subtypes of T.

def:Prop-Set

DEFINITION 2.20.7. The type of types that are propositions and the type of types that are sets are defined as:

$$\text{Prop}_{\mathcal{U}} :\equiv \sum_{X:\mathcal{U}} \text{isProp}(X) \quad \text{and} \quad \text{Set}_{\mathcal{U}} :\equiv \sum_{X:\mathcal{U}} \text{isSet}(X).$$

Both $\text{Prop}_{\mathcal{U}}$ and $\text{Set}_{\mathcal{U}}$ are subtypes of $\mathcal{U}$, and both are types in a universe one higher than $\mathcal{U}$. ⌟

When we don't care about the precise universe $\mathcal{U}$, we'll leave it out from the notation, and just write Prop and Set.

Following Remark 2.20.5, if we have a type A for which we know that it is a proposition or a set, we write also $A : \text{Prop}$ or $A : \text{Set}$, respectively.

def:decidable-set

DEFINITION 2.20.8. A type A is called a *decidable set* if the identity type $x \xrightarrow{=} y$ is a decidable proposition for all $x, y : A$. ⌟

Note the slight subtlety of this definition together with Definition 2.18.1: Any proposition has decidable identity types (since each instance is contractible) and is thus a *decidable set*, even though it may not be a *decidable as a proposition*.

The way we phrased this definition, it builds in the condition that A is a set. The following celebrated and useful theorem states that this is unnecessary.

thm:hedberg

THEOREM 2.20.9 (Hedberg). *Any type A for which we have a function of type $\prod_{x,y:A}\big(x \xrightarrow{=} y \amalg \neg(x \xrightarrow{=} y)\big)$ is a decidable set.*

For a proof see Theorem 7.2.5 of the HoTT Book[60].

[60] Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*.

2.21 *Pointed types*

sec:pointedtypes

Sometimes we need to equip types with additional structure that cannot be expressed by a proposition such as isProp(X) and isSet(X) above. Therefore the following is *not* a subtype of $\mathcal{U}$.

def:pointedtypes

Definition 2.21.1. A *pointed type* is a pair (A, a) where A is a type and a is an element of A. The *type of pointed types* is

$$\mathcal{U}_* :\equiv \sum_{A:\mathcal{U}} A.$$

Given a type A we let A_+ be the pointed type you get by adding a default element: $A_+ :\equiv (A \amalg \mathrm{True}, \mathrm{inr}_{\mathrm{triv}})$. Given a pointed type $X \equiv (A, a)$, the *underlying type* is $X_\div :\equiv A$, and the *base point* is $\mathrm{pt}_X :\equiv a$, so that $X \equiv (X_\div, \mathrm{pt}_X)$.

Let $X :\equiv (A, a)$ and $Y :\equiv (B, b)$ be pointed types. Define the map $\mathrm{ev}_a : (A \to B) \to B$ by $\mathrm{ev}_a(f) :\equiv f(a)$. Then the fiber of ev_a at b is the type $\mathrm{ev}_a^{-1} \equiv \sum_{f:A\to B}(b \xrightarrow{=} f(a))$. The latter type is also called the type of *pointed functions* from X to Y and denoted by $X \to_* Y$. In the notation above

$$(X \to_* Y) \equiv \sum_{f:X_\div \to Y_\div} (\mathrm{pt}_Y \xrightarrow{=} f(\mathrm{pt}_X)).$$

If Z is also a pointed type, and we have pointed functions $(f, f_0) : X \to_* Y$ and $(g, g_0) : Y \to_* Z$, then their composition $(g, g_0)(f, f_0) : X \to_* Z$ is defined as the pair $(gf, g(f_0)g_0)$. See the diagram below.

$$\begin{tikzcd}
 & \mathrm{pt}_Y \arrow[r, "f_0", "="'] \arrow[d, mapsto, "g"] & f(\mathrm{pt}_X) \arrow[d, mapsto, "g"] \\
\mathrm{pt}_Z \arrow[r, "g_0", "="'] & g(\mathrm{pt}_Y) \arrow[r, "g(f_0)", "="'] & g(f(\mathrm{pt}_X))
\end{tikzcd}$$

We may also use the notation $(g, g_0) \circ (f, f_0)$ for the composition. ⌟

def:pointedidentity

Definition 2.21.2. If $X \equiv (A, a)$ is a pointed type, then we define the *pointed identity map* $\mathrm{id}_X : X \to_* X$ by setting $\mathrm{id}_X :\equiv (\mathrm{id}_A, \mathrm{refl}_a)$. ⌟

If X is a pointed type, then $X_\div$ is a type, but X itself is *not* a type. It is therefore unambiguous, and quite convenient, to write $x : X$ for $x : X_\div$, and $X \to \mathcal{U}$ for $X_\div \to \mathcal{U}$. We may also tacitly coerce $f : X \to_* Y$ to $f : X_\div \to Y_\div$.

xca:plusforgetadjoint

Exercise 2.21.3. If A is a type and B is a pointed type, prove that $A \to B_\div$ is equivalent to $A_+ \to_* B$. ⌟

xca:freemaps

Exercise 2.21.4. Let A be a pointed type and B a type. Show that $\sum_{b:B}(A \to_* (B, b))$ and $(A_\div \to B)$ are equivalent. ⌟

2.22 *Operations that produce sets*

sec:operations-on-sets

lem:Set-is-groupoid

Lemma 2.22.1. *If X and Y are sets, then $X \xrightarrow{=} Y$ is a set. In other words,* Set *is a groupoid.*

Proof. By univalence, $(X \xrightarrow{=} Y) \simeq (X \simeq Y) \equiv \sum_{f:X\to Y} \mathrm{isEquiv}(f)$. Since X and Y are sets, so is $X \to Y$ by Lemma 2.15.5. Moreover, isEquiv(f) is a proposition by Lemma 2.15.7. It follows by Corollary 2.20.4 that $X \xrightarrow{=} Y$ is a set. □

One may wonder whether $\mathbb{N}$ as defined in Section 2.12 is a set. The answer is yes, but it is harder to prove than one would think. In fact we have the following theorem.

thm:isset-inductive-types

THEOREM 2.22.2. *All inductive types in Section 2.12 are sets if all constituent types are sets.*

Proof. We only do the case of $\mathbb{N}$ and leave the other cases to the reader (cf. Exercise 2.22.3). The following proof is due to Simon Huber. We have to prove that $n \xrightarrow{=} m$ is a proposition for all $n, m : \mathbb{N}$, i.e., $p \xrightarrow{=} q$ for all $n, m : \mathbb{N}$ and $p, q : n \xrightarrow{=} m$. By induction on q it suffices to prove $p \xrightarrow{=} \mathrm{refl}_n$ for all $p : n \xrightarrow{=} n$. Note that we can not simply prove this by induction on p. Instead we first prove an inversion principle for identifications in $\mathbb{N}$ as follows. We define a type $T(n,m,p)$ for $n, m : \mathbb{N}$ and $p : n \xrightarrow{=} m$ by induction on n and m:

$$T(0,0,p) :\equiv (p \xrightarrow{=} \mathrm{refl}_0) \quad \text{and} \quad T(\mathrm{succ}(n), \mathrm{succ}(m), p) :\equiv \sum_{q : n \xrightarrow{=} m} p \xrightarrow{=} \mathrm{ap}_S\, q,$$

and for the other cases the choice does not matter, say $T(0, \mathrm{succ}(m), p) :\equiv T(\mathrm{succ}(n), 0, p) :\equiv \emptyset$. Next we prove $T(n,m,p)$ for all n, m, and p by induction on p, leaving us with $T(n, n, \mathrm{refl}_n)$ for all $n : \mathbb{N}$, which we in turn prove by distinguishing cases on n. Both the case for 0 and for $\mathrm{succ}(n)$ hold by reflexivity, where in the successor case we use refl_n for q and note that $\mathrm{ap}_S\, \mathrm{refl}_n \equiv \mathrm{refl}_{\mathrm{succ}(n)}$.

We can now prove $p \xrightarrow{=} \mathrm{refl}_n$ for all $p : n \xrightarrow{=} n$ by induction on n. In the base case this is simply $T(0,0,p)$. And for the case $\mathrm{succ}(n)$ we get from $T(\mathrm{succ}(n), \mathrm{succ}(n), p)$ that $p \xrightarrow{=} \mathrm{ap}_S\, q$ for some $q : n \xrightarrow{=} n$. By induction hypothesis we have $e : q \xrightarrow{=} \mathrm{refl}_n$ and thus also

$$p \xrightarrow{=} \mathrm{ap}_S\, q \xrightarrow{=} \mathrm{ap}_S\, \mathrm{refl}_n \equiv \mathrm{refl}_{S(n)}$$

using $\mathrm{ap}_{\mathrm{ap}_S}\, e$, concluding the proof. □

xca:set-sum

EXERCISE 2.22.3. Show that $X \amalg Y$ is a set if X and Y are sets. ⌟

Recall that propositional truncation is turning any type into a proposition by adding identifications of any two elements. Likewise, there is a operation turning any type into a set by adding (higher) identifications of any two identifications of any two elements. The latter operation is called set truncation. It is yet another example of a higher-inductive type.

def:set-truncation

DEFINITION 2.22.4. Let T be a type. The *set truncation* of T is a type $\|T\|_0$ defined by the following constructors:

(1) an *element* $|t|_0 : \|T\|_0$ for all $t : T$;

(2) a *identification* $p \xrightarrow{=} q$ for all $x, y : \|T\|_0$ and $p, q : x \xrightarrow{=} y$.

The (unnamed) second constructor ensures that $\|T\|_0$ is a set. The induction principle states that, for any family of sets $S(x)$ defined for each $x : \|T\|_0$, in order to define a function $f : \prod_{x : \|T\|_0} S(x)$, it suffices to give a function $g : \prod_{t:T} S(|t|_0)$. Computationally, we get $f(|t|_0) \equiv g(t)$ for all $t : T$. ⌟

[61] Lemma 7.3.12[62] gives an equivalence from $|t|_0 \xrightarrow{=} |t'|_0$ to $\|t \xrightarrow{=} t'\|$ for all $t, t : T$.

[62] Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*.

In the non-dependent case we get that for any set S and any function $g : T \to S$ there is a (unique) function $f : \|T\|_0 \to S$ satisfying $f(|t|_0) \equiv g(t)$ for all $t : T$.[61] A consequence of this recursion principle is that, for any set S, precomposition with $|\_|_0$ is an equivalence

$$(\|T\|_0 \to S) \quad \to \quad (T \to S).$$

This is called *the universal property of set truncation*.[63]

[63]More generally, there are operations turning any type into an n-type, satisfying a similar universal property as propositional truncation and set truncation. We denote these operations by $\|\_\|_n$ with corresponding constructor $|\_|_n$. Propositional truncation $\|\_\|$ can thus also be denoted as $\|\_\|_{-1}$. Sometimes it is convenient to consider contractible types as -2-types, with constant truncation operator $\|T\|_{-2} :\equiv \mathrm{True}$ and constructor $|t|_{-2} :\equiv \mathrm{triv}$.

xca:sum-of-conn-components

Exercise 2.22.5. Let A be a type. Define for every element $z : \|A\|_0$ the connected component corresponding to z, $A_{(z)}$, a subtype of A, such that for $a : A$, you recover the notion from Definition 2.16.8: $A_{(|a|_0)} \equiv A_{(a)}$.[64]

[64]*Hint*: Use a map $\|a \xrightarrow{=} \_\| : A \to \mathrm{Prop}$ and the fact that the universe of propositions is a set.

Prove that the set truncation map $|\_|_0 : A \to \|A\|_0$ in this way exhibits A as the sum of its connected components, parametrized by $\|A\|_0$:

$$A \simeq \sum_{z : \|A\|_0} A_{(z)}. \quad \lrcorner$$

2.22.6 *Weakly constant maps*

The universal property of the propositional truncation, Definition 2.16.1, only applies directly to construct elements of *propositions* (that is, to prove them). Here we discuss how we can construct elements of *sets*.

Definition 2.22.7. A map $f : A \to B$ is *weakly constant* if $f(x) \xrightarrow{=} f(x')$ for all $x, x' : A$. $\lrcorner$

This is in contrast to a *constant* map, which can be identified with one of the form $x \mapsto b$ for some $b : B$. Any constant map is indeed weakly constant. Note also that when B is a set, weak constancy of $f : A \to B$ is a proposition.

thm:wconstant-elim

Theorem 2.22.8. *If $f : A \to B$ is a weakly constant map, and B is a set, then there is an induced map $g : \|A\| \to B$ such that $g(|x|) \equiv f(x)$ for all $x : A$.*

Proof. Consider the image factorization $A \xrightarrow{p} \mathrm{im}(f) \xrightarrow{i} B$ of f, where $p(x) :\equiv (f(x), |(x, \mathrm{refl}_{f(x)})|)$ and $i(y, !) :\equiv y$.

The key point is that $\mathrm{im}(f)$ is a proposition: Let $(y_1, z_1), (y_2, z_2) : \mathrm{im}(f)$. Since B is a set, the type $y_1 \xrightarrow{=} y_2$ is a proposition. Hence we may hypothesize (by induction on z_i) that we have $x_1, x_2 : A$ with $f(x_i) \xrightarrow{=} y_i$ for $i = 1, 2$. By concatenation, we get $y_1 \xrightarrow{=} f(x_1) \xrightarrow{=} f(x_2) \xrightarrow{=} y_2$ and hence $(y_1, z_1) \xrightarrow{=} (y_2, z_2)$.

Thus, by the universal property of the truncation, we get $g' : \|A\| \to \mathrm{im}(f)$ such that $g'(|x|) \equiv p(x) \equiv (f(x), |(x, \mathrm{refl}_{f(x)})|)$. Composing with i we get $g :\equiv i \circ g' : \|A\| \to B$ with $g(|x|) :\equiv f(x)$, as desired. □

2.22.9 *Set quotients*

def:quotient-set

Definition 2.22.10. Given a set A and an equivalence relation[65] $R : A \to A \to \mathrm{Prop}$, we define the *quotient set*[66] A/R as the image of the map $R : A \to (A \to \mathrm{Prop})$. For $a : A$ we define $[a] :\equiv (R(a), |(a, \mathrm{refl}_{R(a)})|)$ in A/R; $[a]$ is called the *equivalence class containing a*.[67] $\lrcorner$

[65]Recall that an *equivalence relation* is one that is (1) *reflexive*: $R(x,x)$, (2) *symmetric*: $R(x,y) \to R(y,x)$, and (3) *transitive*: $R(x,y) \to R(y,z) \to R(x,z)$.

[66]We may wonder about the universe level of A/R, assuming $A : \mathcal{U}$ and $R : A \to A \to \mathrm{Prop}_{\mathcal{U}}$. By the Replacement Principle 2.19.4, A/R is essentially $\mathcal{U}$-small, since $A \to \mathrm{Prop}_{\mathcal{U}}$ is locally $\mathcal{U}$-small. Alternatively, we could use Propositional Resizing Principle 2.18.6 to push the values of R into a lower universe.

[67]We recall the convention to use $[a] \equiv (R(a), !)$ also to denote its first component, that is, to use $[a]$ and $R(a)$ interchangeably. The way in which $[a]$ contains a is by observing $R(a) : A \to \mathrm{Prop}$ and $(a, !) : \sum_{x:A} R(a,x)$, by $R(a,a)$.

Any element of the image of R is an equivalence class: a subset P of A for which there exists $a : A$ such that $P(x)$ holds if and only if $R(a,x)$ holds.

In the following proofs we frequently use Exercise 2.17.13.

lem:equiv-class-prop

Lemma 2.22.11. *For any equivalence class $P:A/R$ and $a:A$, $P \overset{=}{\to} [a]$ is inhabited if and only if $P(a)$ holds.*

Proof. Assume we have an identification of type $P \overset{=}{\to} [a]$. Then $P(x)$ is equivalent to $R(a,x)$ for all $x:A$. Take $x :\equiv a$ and use reflexivity $R(a,a)$ to conclude $P(a)$.

Conversely, assume $P(a)$, and let $x:A$ be given. To prove the proposition $P(x) \simeq R(a,x)$ we may assume that $P \equiv [b]$ for some $b:A$. Then $P(x) \equiv R(b,x)$, and we need to show $R(b,x) \simeq R(a,x)$. This follows from $P(a) \equiv R(b,a)$ using symmetry and transitivity. □

thm:quotient-property

Theorem 2.22.12. *We have $([x] \overset{=}{\to} [x']) \simeq R(x,x')$ for all $x,x':A$. Also, for any* set *B, a function $f:A \to B$ factors uniquely through the map $[\_]:A \to A/R$ if $f(x) \overset{=}{\to} f(x')$ for all $x,x':A$ with $R(x,x')$. Indeed we get a map $\bar{f}:A/R \to B$ with $\bar{f}([x]) \equiv f(x)$ for all $x:A$.*

Proof. For the first part we use Lemma 2.22.11 applied to $P_x :\equiv [x]$ and x'.

Now let B be a set and let $f:A \to B$ a function satisfying $f(x) \overset{=}{\to} f(x')$ for all $x,x':A$ with $R(x,x')$.

Uniqueness: If g,h are extensions of f through $[\_]$, then for any $z:A/R$, the type $g(z) \overset{=}{\to} h(z)$ is a proposition since B is a set, so we may assume $z \equiv [x]$ for some $x:A$. Then $g([x]) \overset{=}{\to} f(x) \overset{=}{\to} h([x])$, as desired.

Existence: Let $z \equiv (P,!):A/R$. To define the image of z in B, using the truth of the proposition $\exists_{x:A}(P \overset{=}{\to} [x])$, it suffices by Theorem 2.22.8 to give a weakly constant map $\sum_{x:A}(P \overset{=}{\to} [x]) \to B$, and $f \circ \mathrm{fst}$ does the trick.

Now we check the definitional equality: As an element of A/R, equivalence class $[x]$ is accompanied by the witness $|(x,\mathrm{refl}_{[x]})|:\exists_{y:A}([x] \overset{=}{\to} [y])$. By Theorem 2.22.8, this is mapped, by definition, to $(f\circ\mathrm{fst})(x,\mathrm{refl}_{[x]}) \equiv f(x)$, as desired. □

rem:set-trunc-as-quotient

Remark 2.22.13. We can use set quotients to give an alternative definition of the set truncation $\|A\|_0$ of a type A. Consider the relation $R:A \to A \to \mathrm{Prop}$ given by $R(x,y) :\equiv \|x \overset{=}{\to} y\|$. This is easily seen to be an equivalence relation, using the groupoid structure of identity types. Hence we get a quotient set A/R that satisfies $(|x|_0 \overset{=}{\to} |y|_0) \simeq x = y$, where we write $|\_|_0$ for the equivalence classes. Furthermore, Theorem 2.22.12 implies that A/R satisfies the recursion principle of Definition 2.22.4: If S is a set, and $g:A \to S$ is any function, then $g(x) \overset{=}{\to} g(y)$ holds whenever $x = y$ by the elimination principle of the propositional truncation, and hence we get a function $f:A/R \to S$ satisfying $f(|x|_0) \equiv g(x)$ for all $x:A$, as desired.[68] ⌟

[68] Expanding the definitions, this means that we can take the 0-truncation $\|A\|_0$ of $A:\mathcal{U}$ to be the $\mathcal{U}$-small image of the (-1)-truncated identity relation $A \to (A \to \mathrm{Prop}_{\mathcal{U}})$. Similarly, we can recursively construct the $(n+1)$-truncation by taking the $\mathcal{U}$-small image of the n-truncated identity relation $A \to (A \to \sum_{X:\mathcal{U}} \mathrm{is}n\mathrm{Type})$.

sec:more-on-N

2.23 *More on natural numbers*

A useful function $\mathbb{N} \to \mathbb{N}$ is the *predecessor* pred defined by $\mathrm{pred}(0) :\equiv 0$ and $\mathrm{pred}(\mathrm{succ}(n)) :\equiv n$. Elementary properties of addition, multiplication and predecessor can be proved in type theory in the usual way. We freely use them, sometimes even in definitions, leaving most of the proofs/constructions to the reader.

def:orderonN

DEFINITION 2.23.1. Let $n, m : \mathbb{N}$. We say that m is less than or equal to n, and write $m \leq n$, if there is a $k : \mathbb{N}$ such that $k + m \overset{=}{\to} n$. Such a k is unique, and if it is not 0, we say that m is less than n, denoted by $m < n$. Both $m \leq n$ and $m < n$ are propositions for all $n, m : \mathbb{N}$. ⌟

xca:try-your-luck-N

EXERCISE 2.23.2. Try your luck in type theory proving any of the following. The successor function satisfies $(\mathrm{succ}(n) \overset{=}{\to} \mathrm{succ}(m)) \simeq (n \overset{=}{\to} m)$. The functions $+$ and $\cdot$ are commutative and associative, $\cdot$ distributes over $+$. The relations $\leq$ and $<$ are transitive and preserved under $+$; $\leq$ also under $\cdot$. We have $(m \leq n) \simeq ((m < n) \amalg (m \overset{=}{\to} n))$ (so $\leq$ is reflexive). Furthermore, $((m \leq n) \times (n \leq m)) \simeq (m \overset{=}{\to} n)$, and $\neg((m < n) \times (n < m))$ (so $<$ is irreflexive). ⌟

We can prove the following lemma by double induction.

lem:dec-eq+order-N

LEMMA 2.23.3. *The relations $\overset{=}{\to}$, $\leq$ and $<$ on $\mathbb{N}$ are decidable.*

By Hedberg's Theorem 2.20.9, we get an alternate proof that $\mathbb{N}$ is a set.

We will now prove an important property of $\mathbb{N}$, called the *least number principle for decidable, non-empty subsets of* $\mathbb{N}$. We give some more details of the proof, since they illustrate an aspect of type theory that has not been very prominent up to know, namely the close connection between proving and computing.

def:Nwellordered

CONSTRUCTION 2.23.4. *Let $P(n)$ be a proposition for all natural numbers n. Define the type $P_{\min}(n)$ expressing that n is the smallest natural number such that $P(n)$:*

$$P_{\min}(n) :\equiv P(n) \times \prod_{m : \mathbb{N}} (P(m) \to n \leq m)$$

Then we seek a function

{eqn:min}

$$\min(P) : \prod_{n : \mathbb{N}} (P(n) \amalg \neg P(n)) \to \exists_{n : \mathbb{N}} P(n) \to \sum_{n : \mathbb{N}} P_{\min}(n), \tag{2.23.1}$$

computing a minimal witness for P from evidence that P is decidable and that a witness exists.

Implementation of Construction 2.23.4. First note that $P_{\min}(n)$ is a proposition, and that all n such that $P_{\min}(n)$ are equal. Therefore the type $\sum_{n : \mathbb{N}} P_{\min}(n)$ is also a proposition.

Given a function $d(n) : P(n) \amalg \neg P(n)$ deciding $P(n)$ for each $n : \mathbb{N}$, we define a function $\mu_P : \mathbb{N} \to \mathbb{N}$ which, given input n, searches for a $k < n$ such that $P(k)$. If such a k exists, μ_P returns the least such k, otherwise $\mu_P(n) \overset{=}{\to} n$. This is a standard procedure that we will call *bounded search*. The function μ_P is defined by induction, setting $\mu_P(0) :\equiv 0$ and $\mu_P(\mathrm{succ}(n)) :\equiv \mu_P(n)$ if $\mu_P(n) < n$. Otherwise, we set $\mu_P(\mathrm{succ}(n)) :\equiv n$ if $P(n)$, and $\mu_P(\mathrm{succ}(n)) :\equiv \mathrm{succ}(n)$ otherwise, using $d(n)$ to decide, that is, by induction on $d(n) : P(n) \amalg \neg P(n)$. By design, μ_P 'remembers' where it has found the least k (if so). We are now done with the computational part and the rest is a correctness proof.

By induction on $n : \mathbb{N}$ and $d(n) : P(n) \amalg \neg P(n)$ we show

$$\mu_P(n) \leq n \quad \text{and} \quad \mu_P(n) < n \to P(\mu_P(n)).$$

The base case where $n :\equiv 0$ is easy. For the induction step, review the computation of $\mu_P(\mathrm{succ}(n))$. If $\mu_P(\mathrm{succ}(n)) \overset{=}{\to} \mu_P(n)$ since $\mu_P(n) < n$, then we are done by the induction hypothesis. Otherwise, either

$\mu_P(\mathrm{succ}(n)) \stackrel{=}{\rightarrow} n$ and $P(n)$, or $\mu_P(\mathrm{succ}(n)) \stackrel{=}{\rightarrow} \mathrm{succ}(n)$. In both cases we are done.

Also by induction on $n : \mathbb{N}$ and $d(n) : P(n) \amalg \neg P(n)$ we show

$$P(m) \to \mu_P(n) \leq m, \text{ for all } m \text{ in } \mathbb{N}.$$

The base case $n :\equiv 0$ holds since $\mu_P(0) \stackrel{=}{\rightarrow} 0$. For the induction step, assume $P(m) \to \mu_P(n) \leq m$ for all m (IH). Let $m : \mathbb{N}$ and assume $P(m)$. We have to prove $\mu_P(\mathrm{succ}(n)) \leq m$. If $\mu_P(\mathrm{succ}(n)) \stackrel{=}{\rightarrow} \mu_P(n)$ we are done by IH. Otherwise we have $\mu_P(n) \stackrel{=}{\rightarrow} n$ and $\mu_P(\mathrm{succ}(n)) \stackrel{=}{\rightarrow} \mathrm{succ}(n)$ and $\neg P(n)$. Then $\mu_P(n) \leq m$ by IH, and $n \neq m$, so $\mu_P(\mathrm{succ}(n)) \leq m$.

By contraposition we get from the previous result

$$\mu_P(n) \stackrel{=}{\rightarrow} n \to \neg P(m), \text{ for all } m < n.$$

Note that there may not be any n such that $P(n)$; the best we can do is to prove

$$P(n) \to P_{\min}(\mu_P(\mathrm{succ}(n)))$$

by combining previous results. Assume $P(n)$. Then $\mu_P(\mathrm{succ}(n)) \leq n < \mathrm{succ}(n)$, so that $P(\mu_P(\mathrm{succ}(n)))$. Moreover, $P(m) \to \mu_P(\mathrm{succ}(n)) \leq m$ for all m in $\mathbb{N}$. Hence $P_{\min}(\mu_P(\mathrm{succ}(n)))$.

Since $\sum_{n:\mathbb{N}} P_{\min}(n)$ is a proposition, we obtain the required function by the induction principle for propositional truncation, Definition 2.16.1:

$$\min(P) : \prod_{n:\mathbb{N}} (P(n) \amalg \neg P(n)) \to \Big\| \sum_{n:\mathbb{N}} P(n) \Big\| \to \sum_{n:\mathbb{N}} P_{\min}(n). \qquad \square$$

rem:computations-can-decide

REMARK 2.23.5. In the interest of readability, we do not always make the use of witnesses of decidability in computations explicit. A typical example is the case distinction on $\mu_P(n) < n$ in Construction 2.23.4 above. This remark applies to all sets and decidable relations on them. We shall immediately put this convention to good use in the proof of a form of the so-called *Pigeonhole Principle* (PHP). ⌟

lem:PHP

LEMMA 2.23.6. *For all $N : \mathbb{N}$ and $f : \mathbb{N} \to \mathbb{N}$ such that $f(n) < N$ for all $n < N+1$, there exist $m < n < N+1$ such that $f(n) \stackrel{=}{\rightarrow} f(m)$.*

Proof. By induction on N. In the base case $N \stackrel{=}{\rightarrow} 0$ there is nothing to do. For the induction case $N+1$, assume the lemma proved for N (induction hypothesis, IH, for all f). Let f be such that $f(n) < N+1$ for all $n < N+2$. The idea of the proof is to search for an $n < N+1$ such that $P(n) :\equiv (f(n) \stackrel{=}{\rightarrow} N)$, by computing $\mu_P(N+1)$ as in Construction 2.23.4. If $\mu_P(N+1) \stackrel{=}{\rightarrow} N+1$, that is, $f(n) < N$ for all $n < N+1$, then we are done by IH. Assume $\mu_P(N+1) < N+1$, so $f(\mu_P(N+1)) \stackrel{=}{\rightarrow} N$. If also $f(N+1) \stackrel{=}{\rightarrow} N$ then we are done. If $f(N+1) < N$, then we define g by $g(n) \stackrel{=}{\rightarrow} f(N+1)$ if $f(n) \stackrel{=}{\rightarrow} N$, and $g(n) \stackrel{=}{\rightarrow} f(n)$ otherwise. Then IH applies to g, and we get $m < n < N+1$ with $g(n) \stackrel{=}{\rightarrow} g(m)$. If $f(n) \stackrel{=}{\rightarrow} f(m)$ we are of course done. Otherwise, $f(n), f(m)$ cannot both be smaller than N, as $g(n) \stackrel{=}{\rightarrow} g(m)$. In both remaining cases, $f(n) \stackrel{=}{\rightarrow} g(n) \stackrel{=}{\rightarrow} g(m) \stackrel{=}{\rightarrow} f(N+1)$ and $f(N+1) \stackrel{=}{\rightarrow} g(n) \stackrel{=}{\rightarrow} g(m) \stackrel{=}{\rightarrow} f(m)$, we are done. $\square$

We can now rule out the existence of equivalences between finite sets of different size.

cor:Fin-n-injective

COROLLARY 2.23.7. *If $m < n$, then $(\sum_{k:\mathbb{N}} k < m) \neq (\sum_{k:\mathbb{N}} k < n)$.*

Another application of Construction 2.23.4 is a short proof of Euclidean division.

lem:euclid-div

LEMMA 2.23.8. *For all $n, m : \mathbb{N}$ with $m > 0$ there exist unique $q, r : \mathbb{N}$ such that $r < m$ and $n \overset{=}{\to} qm + r$.*

Proof. Define $P(k) :\equiv (n \leq km)$. Since $m > 0$ we have $P(n)$. Now set $k :\equiv \mu_P(n)$ as in Construction 2.23.4. If $n \overset{=}{\to} km$ and we set $q :\equiv k$ and $r :\equiv 0$. If $n < km$, then $k > 0$ and we set $q :\equiv k - 1$. By minimality we have $qm < n < km$ and hence $n \overset{=}{\to} qm + r$ for some $r < m$. □

2.24 *The type of finite types*

sec:typeFin

Recall from Section 2.12.1 the types False, True and Bool containing zero, one and two elements, respectively. We now define generally the type of n elements for any $n : \mathbb{N}$.

def:finiteset

DEFINITION 2.24.1. For any type X define $\mathrm{succ}(X) :\equiv X \amalg \mathrm{True}$. Define inductively the type family $F(n)$, for each $n : \mathbb{N}$, by setting $F(0) :\equiv \emptyset$ and $F(\mathrm{succ}(n)) :\equiv \mathrm{succ}(F(n))$. Now abbreviate $\mathbb{n} :\equiv F(n)$. The type $\mathbb{n}$ is called the type with n elements, and we denote its elements by $0, 1, \ldots, n-1$ rather than by the corresponding expressions using inl and inr.

We also define $\mathbb{m} :\equiv F(m)$ for a natural number m, $\mathbb{0} :\equiv F(0)$, $\mathbb{1} :\equiv F(1)$, and $\mathbb{2} :\equiv F(2)$. ⌟

xca:finite-types

EXERCISE 2.24.2.

(1) Denote in full all elements of $\mathbb{0}$, $\mathbb{1}$, and $\mathbb{2}$.

(2) Show (using univalence) that $\mathbb{1} \overset{=}{\to} \mathrm{True}$, $\mathbb{2} \overset{=}{\to} \mathrm{Bool}$.

(3) Show (using univalence) that $\mathbb{n} \overset{=}{\to} \sum_{k:\mathbb{N}} k < n$ for all $n : \mathbb{N}$.

(4) Show that $m = n$ if $\mathbb{m} = \mathbb{n}$. ⌟

def:is-finite

DEFINITION 2.24.3. Given a type X, we define the proposition

$$\mathrm{isFinSet}(X) :\equiv \exists_{n:\mathbb{N}}(X \overset{=}{\to} \mathbb{n})$$

to express that X *is a finite set*.[69] ⌟

[69]When moving beyond sets, there are two different ways in which a type can be finite: an *additive* way and a *multiplicative* way, but it would take us too far afield to define these notions here.

lem:maxonefinitetype

LEMMA 2.24.4.

(1) *$\sum_{n:\mathbb{N}} X = \mathbb{n}$ is a proposition, for all types X.*

(2) *$\sum_{X:\mathcal{U}} \sum_{n:\mathbb{N}} X = \mathbb{n} \overset{=}{\to} \sum_{X:\mathcal{U}} \mathrm{isFinSet}(X)$.*

Proof. (1) Assume $(n, p), (m, q) : \sum_{n:\mathbb{N}} X = \mathbb{n}$. Then we have $\mathbb{n} = \mathbb{m}$, so $n = m$ by Exercise 2.24.2. But $\mathbb{N}$ is a set by Theorem 2.22.2, so $n = m \overset{=}{\to} (n \overset{=}{\to} m)$. It follows that $(n, p) \overset{=}{\to} (m, q)$.

(2) Follows from $\sum_{n:\mathbb{N}} X = \mathbb{n} = \|\sum_{n:\mathbb{N}} X \overset{=}{\to} \mathbb{n}\|$, which is easily proved by giving functions in both directions and using the univalence axiom. □

The lemma above remains true if X ranges over Set. If a set S is in the same component in Set[70] as $\mathbb{n}$ we say that *S has cardinality n* or that *the cardinality of S is n*.

[70]Here it doesn't matter whether we say Set or $\mathcal{U}$, since any finite set is a set. Hence we also have $\mathrm{FinSet}_n \equiv \mathrm{Set}_{(\mathbb{n})} \overset{=}{\to} \mathrm{FinSet}_{(\mathbb{n})} \overset{=}{\to} \mathcal{U}_{(\mathbb{n})}$.

def:groupoidFin

Definition 2.24.5. The *groupoid of finite sets* is defined by

$$\mathrm{FinSet} :\equiv \sum_{S:\mathrm{Set}} \mathrm{isFinSet}(S).$$

For $n:\mathbb{N}$, the *groupoid of sets of cardinality n* is defined by

$$\mathrm{FinSet}_n :\equiv \sum_{S:\mathrm{Set}} S = \mathbb{m}. \quad ⌟$$

Observe that $\mathrm{FinSet}_0 \xrightarrow{=} \mathrm{FinSet}_1 \xrightarrow{=} \mathbb{1}$ and $\mathrm{FinSet} \xrightarrow{=} \sum_{n:\mathbb{N}} \mathrm{FinSet}_n$ by Lemma 2.24.4.

Note that being a finite set implies being a set, and hence $\mathrm{FinSet} \xrightarrow{=} \sum_{X:\mathcal{U}} \mathrm{isFinSet}(X)$. Also, FinSet is the image of the map $F:\mathbb{N} \to \mathcal{U}$ from Definition 2.24.1, and is hence essentially $\mathcal{U}$-small (for any universe $\mathcal{U}$), by Principle 2.19.4, Item (P1), and our assumption that $\mathcal{U}_0$ is the smallest universe.

2.25 *Type families and maps*

sec:typefam

There is a natural equivalence between maps into a type A and type families parametrized by A. The key idea is that the fibers of a map form a type family. We will elaborate this idea and some variations.

lem:fst-fiber(a)=B(a)

Lemma 2.25.1. *Let $A:\mathcal{U}$ and $B:A\to\mathcal{U}$. Recall the function* $\mathrm{fst}:(\sum_{a:A} B(a)) \to A$*. Then $e_a : B(a) \to \mathrm{fst}^{-1}(a)$ defined by $e_a(b) :\equiv ((a,b),\mathrm{refl}_a)$ is an equivalence, for all $a:A$.*

Proof. Note that $\mathrm{fst}(x,b) \equiv x$ and that $a \xrightarrow{=} x$ does not depend on b. Hence $\mathrm{fst}^{-1}(a) \simeq \sum_{x:A}(B(x)\times(a \xrightarrow{=} x))$ via rearranging brackets. Applying Corollary 2.9.11 leads indeed to the equivalence e_a. □

lem:sum-of-fibers

Lemma 2.25.2. *Let $A,B:\mathcal{U}$ and $f:B\to A$. Then $e:B\to\sum_{a:A} f^{-1}(a)$ defined by $e(b):\equiv(f(b),b,\mathrm{refl}_{f(b)})$ is an equivalence.*

Proof. Define $e^{-1}:\sum_{a:A} f^{-1}(a)\to B$ by $e(a,b,p):\equiv b$. Then $e^{-1}(e(b))\equiv b$ for all $b:B$. Let $a:A$, $b:B$ and $p:f(b)\xrightarrow{=} a$. Then $e(e^{-1}(a,b,p))\equiv (f(b),b,\mathrm{refl}_{f(b)})$. We have to prove $(f(b),b,\mathrm{refl}_{f(b)}) \xrightarrow{=} (a,b,p)$. We use p as identification of the first components, and refl_b as identification of the second components (whose type is constant). For the third component we use that the transport of $\mathrm{refl}_{f(b)}$ along p in the type family $(f(b)\xrightarrow{=} \_)$ is indeed equal to p itself by Exercise 2.14.4(2). Now apply Lemma 2.9.9. □

If f above is an injection, then $\sum_{a:A} f^{-1}(a)$ is a subtype of A, and B is a n-type if A is a n-type by Corollary 2.20.4.

lem:typefamiliesandfibrations

Lemma 2.25.3. *Let A be a type. Then*

$$\mathrm{preim} \;:\; \sum_{B:\mathcal{U}}(B\to A) \quad\to\quad (A\to\mathcal{U})$$

given by $\mathrm{preim}(B,f)(a):\equiv f^{-1}(a)$ *is an equivalence. An inverse equivalence is given by sending $P:A\to\mathcal{U}$ to* $(\sum_{a:A}P(a),\mathrm{fst})$.

Proof. We apply Lemma 2.9.9, and verify the two conditions. Let $P:A\to\mathcal{U}$. We have to prove that $P \xrightarrow{=} \mathrm{preim}(\sum_{a:A}P(a),\mathrm{fst})$. By function extensionality it suffices to prove $\mathrm{preim}(\sum_{a:A}P(a),\mathrm{fst})(a)\equiv \mathrm{fst}^{-1}(a)\xrightarrow{=} P(a)$. This follows directly from Lemma 2.25.1 and the univalence axiom.

Let $f : B \to A$. We have to prove that $(\sum_{a:A} f^{-1}(a), \mathrm{fst}) \overset{=}{\to} (B, f)$. Using the univalence axiom, we get an identification $\bar{e} : \sum_{a:A} f^{-1}(a) \overset{=}{\to} B$, where e is the equivalence from Lemma 2.25.2. Using Lemma 2.10.3, it remains to give an element of the type $\mathrm{fst} \underset{\bar{e}}{\overset{=}{\to}} f$.

As an auxiliary step we note that for any $p : X \overset{=}{\to} Y$ and $g : X \to A$, $h : Y \to A$, the type $g \underset{p}{\overset{=}{\to}} h$ of paths over p is equal to the type $g \overset{=}{\to} h \circ \tilde{p}$, since the two types are definitionally equal for $p \equiv \mathrm{refl}_X$. Applying this here means that we must give an element of $\mathrm{fst} \overset{=}{\to} f \circ \tilde{\bar{e}}$. This in turn means that we must give an element of $\mathrm{fst} \overset{=}{\to} f \circ e$, which follows by function extensionality from the definition of e in Lemma 2.25.2. □

Let A be a type and consider the subuniverse $\mathrm{Prop} \equiv \sum_{X:\mathcal{U}} \mathrm{isProp}(X)$ from Section 2.20. A function $P : A \to \mathrm{Prop}$ can be viewed as a family of propositions: $\mathrm{fst} \circ P : A \to \mathcal{U}$ is a type family, and $\mathrm{snd} \circ P : \prod_{a:A} \mathrm{isProp}(P(a))$ witnesses that each $\mathrm{fst}(P(a))$ is a proposition. The inverse equivalence in Lemma 2.25.3 sends $\mathrm{fst} \circ P$ to

$$\mathrm{fst} : \Big(\sum_{a:A} \mathrm{fst}(P(a))\Big) \to A.$$

All the fibers of this function are propositions by combining $\mathrm{snd} \circ P : \prod_{a:A} \mathrm{isProp}(P(a))$ with Lemma 2.25.1.

Conversely, for a function $f : B \to A$ with proof $g : \prod_{a:A} \mathrm{isProp}(f^{-1}(a))$ that all fibers of f are propositions, we can define $P_f : A \to \mathrm{Prop}$ by setting $P_f(a) :\equiv (f^{-1}(a), g(a))$.

The above argument can be refined for each of $\mathrm{Prop}, \mathrm{Set}, \mathcal{U}_*$ from Section 2.20, and one can prove the following analogues of Lemma 2.25.3.

lem:Prop-Set-pointed-families

lem:Set-families

LEMMA 2.25.4. *Let A be a type. Then we have:*

(1) $(A \to \mathrm{Prop}) \simeq \sum_{B:\mathcal{U}} \sum_{f:B\to A} \prod_{a:A} \mathrm{isProp}(f^{-1}(a))$;

(2) $(A \to \mathrm{Set}) \simeq \sum_{B:\mathcal{U}} \sum_{f:B\to A} \prod_{a:A} \mathrm{isSet}(f^{-1}(a))$;

(3) $(A \to \mathcal{U}_*) \simeq \sum_{B:\mathcal{U}} \sum_{f:B\to A} \prod_{a:A} f^{-1}(a)$. *(Hard!)*

Since Prop is a set, we obtain the following corollary.

cor:Sub_T-is-set

COROLLARY 2.25.5. *Subtypes as in Definition 2.20.2 correspond to predicates and* Sub_T *is a set, for any type T.*

sec:stuff-struct-prop

2.26 *Higher structure: stuff, structure, and properties*

The precise formalization of the intuitive notions of "stuff", "structure", and "properties" was worked out in terms of category theory in *UseNet* discussions between John Baez, Toby Bartels, and James Dolan on `sci.physics.research` in 1998. It was clear that the simplest description was in terms of homotopy types, and hence it's even simpler in type theory. See also Baez and Shulman[71] for further discussion.

Recall from Lemma 2.25.2 that any map $f : B \to A$ can be described as "projecting away" its fibers, by using the equivalence e:

{eq:forget-fibers}

(2.26.1)
$$\begin{tikzcd} B \arrow[rr, "e", "\sim"'] \arrow[dr, "f"'] & & \sum_{a:A} f^{-1}(a) \arrow[dl, "\mathrm{fst}"] \\ & A & \end{tikzcd}$$

We say that f *forgets* these fibers. If A and B are groupoids, these fibers are themselves groupoids, but it can happen that they are sets, propositions, or even contractible. Accordingly, we say that:

- f *forgets at most structure* if all the fibers are sets;
- f *forgets at most properties* if all the fibers are propositions;

[71]John C. Baez and Michael Shulman. "Lectures on n-categories and cohomology". In: *Towards higher categories*. Vol. 152. IMA Vol. Math. Appl. Springer, New York, 2010, pp. 1–68. DOI: 10.1007/978-1-4419-1524-5_1. arXiv: math/0608420.

- f *forgets nothing* if all the fibers are contractible.

Here, the structure and properties in question are *on a* or *of a*, respectively, as captured by the fibers at a, for each $a : A$. Of course, a map forgets properties if and only if it's an injection, and it forgets nothing if and only if it's an equivalence.

Going in the other direction, we say that:

- f *forgets at most n-structure* if all the fibers are n-truncated. If $n \geq 1$, this is therefore a kind of *higher structure*.[72]

[72] We're updating the terminology slightly: In the above references, n-structure is referred to as *n-stuff*, but nowadays the term *higher structure* is more common, so we have renamed n-stuff into *n-structure*.

Thus, an element of a groupoid is 1-structure (this is sometimes informally called *stuff*), while an element of a set is a structure, or 0-structure, while an proof of a proposition is a property, or (-1)-structure.

Looking at (2.26.1) another way, we see that to give an element of b of B lying over a given element $a : A$ amounts to specifying an element on $f^{-1}(a)$, so we say that the elements of B are elements of A *with extra n-structure*, if the fibers $f^{-1}(a)$ are n-truncated.

Refining the usual image and image factorization from Definition 2.17.11 and Exercise 2.17.12 we can factor $f : B \to A$ through first its 0-*image* and then its usual (-1)-image as follows:[73]

[73] Using the general n-truncation, we can define the n-image in a similar way and prove that the n-image factorization is unique. Since the unit type $\mathbb{1}$ is the unique (-2)-type, we have $\|X\|_{-2} \xrightarrow{=} \mathbb{1}$ for any type X.

$$B \xrightarrow{=} \sum_{a:A} f^{-1}(a) \to \sum_{a:A} \|f^{-1}(a)\|_0 \to \sum_{a:A} \|f^{-1}(a)\|_{-1} \to \sum_{a:A} \|f^{-1}(a)\|_{-2} \xrightarrow{=} A.$$

Here, the first map *forgets pure higher structure*, the second map *forgets pure structure*, while the last forgets at most properties (this is the inclusion of the usual image). Of course, each of these maps may happen to forget nothing at all. Saying that the second map forgets *pure* structure indicates that not only are the fibers sets, they are *nonempty* sets, so the structure in question exists, at least. Note also that the fibers of the first map are connected, which indicates that what is forgotten at this step, if anything, is pure higher structure.

exa:stuff-struct-prop

EXAMPLE 2.26.1. Let us look at some examples:

- The first projection $\mathrm{fst} : \mathrm{FinSet} \times \mathrm{FinSet} \to \mathrm{FinSet}$ forgets 1-structure (stuff), namely the second set in the pair.
- The first projection $\mathrm{fst} : \sum_{A:\mathrm{FinSet}} A \to \mathrm{FinSet}$ from the type of pointed finite sets to the type of finite sets forgets structure, namely the structure of a chosen point.
- The inclusion of the type of sets with cardinality n, FinSet_n, into the type of all finite sets, FinSet, forgets properties, namely the property "having cardinality n". ⌟

xca:stuff-struct-prop

EXERCISE 2.26.2. Analyze more examples of maps between groupoids in terms of "what is forgotten". ⌟

xca:0Im-to-Im

EXERCISE 2.26.3. Let $|\_|' : \|f^{-1}(a)\|_0 \to \|f^{-1}(a)\|$ be the map defined by the induction principle in Definition 2.22.4 from $|\_| : f^{-1}(a) \to \|f^{-1}(a)\|$. In the refined image factorization above, the map for the second arrow maps any pair (a, x) with $x : \|f^{-1}(a)\|_0$ to the pair $(a, |x|')$. Show that for any $p : \|f^{-1}(a)\|$ the fiber of the latter map at (a, p) is equivalent to $\|f^{-1}(a)\|_0$. What is forgotten by this map, and what is remembered? ⌟

Recall the (−1-)image factorization and its uniqueness from Exercise 2.17.12. The 0-image factorization and its uniqueness play just as important a role, so we give a full proof.

thm:unique-fact-0image

THEOREM 2.26.4. *Show that the* 0*-image of* $f : A \to B$ *induces a factorization* $f \xrightarrow{=} i \circ p$

$$\begin{tikzcd} A \arrow[rr, "f"] \arrow[dr, "p"'] & & B \\ & \mathrm{im}(f) \arrow[ur, "i"'] & \end{tikzcd}$$

where p is 0*-connected and i is* 0*-truncation, and that each such factorization is equivalent to the* 0*-image factorization.*

2.27 *Higher truncations*

sec:higher-truncations

We've seen the propositional truncation in Section 2.16 and the set truncation in Section 2.22. As mentioned in Remark 2.22.13, it's possible to define the latter in terms of the former by considering the propositional truncation of the identity types of a type A. In this section we want to generalize this to higher truncation levels and show how we can inductively define all the n-truncation operations using propositional truncation combined with the replacement principle, Principle 2.19.4, which is used to stay within a given universe.

def:join-construction-of-truncation

CONSTRUCTION 2.27.1. *For any integer* $n \geq -1$ *we have an* n-truncation operation $\|\_\|_n : \mathcal{U} \to \mathcal{U}$, *along with* unit maps $|\_|_n : A \to \|A\|_n$, *satisfying the following universal property.*

For any n*-type B, precomposition with* $|\_|_n$ *induces an equivalence*:

$$\big(\|A\|_n \to B\big) \xrightarrow{\simeq} (A \to B).$$

Implementation of Construction 2.27.1. We proceed by induction. For $n \equiv -1$, we have this from the higher inductive type definition, Definition 2.16.1, with element constructor $|\_| : A \to \|A\|$.

To go from n to $n+1$, we fix a type $A : \mathcal{U}$ and consider the n-truncated identity relation

$$I_n : A \to \left(A \to \sum_{X:\mathcal{U}} \mathrm{is}n\mathrm{Type}(X)\right), \quad x \mapsto (y \mapsto \|x \xrightarrow{=} y\|_n).$$

Let $\|A\|_{n+1} :\equiv \mathrm{im}(I_n)$ be the usual image of I_n, and let $|\_|_{n+1} : A \to \|A\|_{n+1}$ be the image inclusion, $x \mapsto (\|x \xrightarrow{=} \_\|_n, !)$.

Since the type of n-types is an $(n+1)$-type, $\|A\|_{n+1}$ is an $(n+1)$-type by Lemma 2.15.5. We also note that the map

{eq:trunc-path-eq}

$$\|x \xrightarrow{=} y\|_n \xrightarrow{\simeq} (|x|_{n+1} \xrightarrow{=} |y|_{n+1}), \tag{2.27.1}$$

induced by the universal property of n-truncation, is an equivalence. Indeed, the right-hand side is equivalent to

$$\prod_{z:A}\big(\|x \xrightarrow{=} z\|_n \xrightarrow{\simeq} \|y \xrightarrow{=} z\|_n\big),$$

and we get an inverse by going backwards along this equivalence at $|\mathrm{refl}_y|_n : \|y \xrightarrow{=} y\|_n$.

To prove the universal property, let B be any $(n+1)$-type and $g : A \to B$ any map.

It suffices to show that for any $z : \|A\|_{n+1}$, there is a contractible type of extensions

$$\begin{tikzcd}
{|\_|_{n+1}^{-1}(z)} \arrow[d] \arrow[dr, "g \circ \mathrm{fst}"] & \\
\mathbb{1} \arrow[r, dashed] & B,
\end{tikzcd}$$

since then there's a contractible type of extensions of g to all of $\|A\|_{n+1}$ Since this is a proposition and $|\_|_{n+1}$ is surjective, it suffices to prove this for z of the form $|x|_{n+1}$ with $x : A$. We need to show that the type

$$\prod_{x:A} \sum_{y:B} \prod_{x':A} \big((|x|_{n+1} \xrightarrow{=} |x'|_{n+1}) \to (y \xrightarrow{=} g(x'))\big)$$

is contractible. By the equivalence above, we can rewrite this, first as

$$\prod_{x:A} \sum_{y:B} \prod_{x':A} \big(\|x \xrightarrow{=} x'\|_n \to (y \xrightarrow{=} g(x'))\big),$$

and then, since $y \xrightarrow{=} g(x')$ is an n-type, as

$$\prod_{x:A} \sum_{y:B} \prod_{x':A} \big((x \xrightarrow{=} x') \to (y \xrightarrow{=} g(x'))\big).$$

Now we can contract away x' and the identification $x \xrightarrow{=} x'$, so we're left with

$$\prod_{x:A} \sum_{y:B} (y \xrightarrow{=} g(x')),$$

which is indeed contractible.

Finally, we need to re-size $\|A\|_{n+1}$ to fit in the universe $\mathcal{U}$ that A came from. By (2.27.1), its identity types are essentially $\mathcal{U}$-small by induction hypothesis, so again since $|\_|_{n+1}$ is a surjection from the $\mathcal{U}$-small type A, the replacement principle, Principle 2.19.4, implies that $\|A\|_{n+1}$ is essentially $\mathcal{U}$-small. □

This construction is due to Rijke[74], see also the presentation in his book[75].

[74] Rijke, *The join construction*.

[75] Egbert Rijke. *Introduction to Homotopy Type Theory*. Forthcoming book with CUP. Version from 06/02/22. 2022.

3
The universal symmetry: the circle

cha:circle

An effective principle in mathematics is that when you want to study a certain phenomenon you should search for a single type that captures this phenomenon. Here are two examples:[1]

[1]Notice that these have arrows pointing in different directions: In Item (1) we're mapping *out* of $\mathbb{1}$, while in Item (2) we're mapping *in* to Prop.

it:one-out

(1) The contractible type $\mathbb{1}$ has the property that given any type A a function $\mathbb{1} \to A$ provides exactly the same information as picking an element in A. For, an equivalence from A to $\mathbb{1} \to A$ is provided by the function $a \mapsto (x \mapsto a)$.

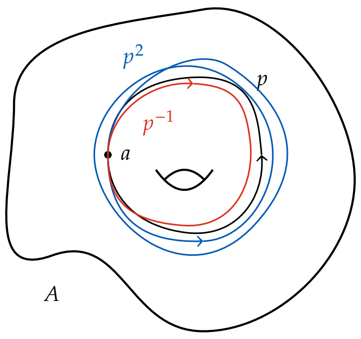
it:prop-in

(2) The type Prop of propositions has the property that given any type A a function $A \to \mathrm{Prop}$ provides exactly the same information as picking a subtype of A.

We are interested in symmetries, and so we should search for a type X which is so that given *any* type A the type of functions $X \to A$ (or $A \to X$, but that's not what we're going to do) picks out exactly the symmetries in A. We will soon see that there is such a type: the circle[2] which is built *exactly* so that this "universality with respect to symmetries" holds. It may be surprising to see how little it takes to define it; especially in hindsight when we eventually discover some of the many uses of the circle.

[2]We call this type the "circle" because it has many properties which are analogues, in our context, of properties of the topological circle $\{ (x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1 \}$. See Appendix B.3 for a discussion of the relationship between topological spaces and types. In the later chapters on geometry we'll return to "real" geometrical circles.

A symmetry in A is an identification $p : a \xrightarrow{=}_A a$ for some $a : A$. Now, we can take any iteration of p (composing p with itself a number of times), and we can consider the inverse p^{-1} and *its* iterations. So, by giving one symmetry we at the same time give a lot of symmetries. For a particular p it may be that not all of the iterations are different, for instance it may be that there is an identification of type $p^2 \xrightarrow{=} p^0$ (as in Exercise 2.13.3), or even more dramatically: if there is an identification of type $p = \mathrm{refl}_a$, then *all* the iterations of p can be identified with each other. However, in general we must be prepared that all the powers of p (positive, 0 and negative) are distinct. Hence, the circle must have a distinct symmetry for every integer. We would have enjoyed defining the integers this way, but being that ideological would be somewhat inefficient. Hence we give a more hands-on approach and define the circle and the integers separately. Thereafter we prove that the type of symmetries in the circle is equivalent to the set of integers.



3.1 *The circle and its universal property*

sec:S1

Propositional truncation from Section 2.16 was the first *higher inductive type*, that is, an inductive type with constructors both for elements and for